

TRƯỜNG ĐẠI HỌC TRÀ VINH
KHOA KỸ THUẬT VÀ CÔNG NGHỆ
BỘ MÔN CÔNG NGHỆ THÔNG TIN



TÀI LIỆU GIẢNG DẠY

MÔN CẤU TRÚC DỮ LIỆU

VÀ GIẢI THUẬT 1

GV biên soạn: *Lê Minh Tự*
 Phan Quốc Nghĩa

Trà Vinh, 2013

Lưu hành nội bộ

MỤC LỤC

CHƯƠNG 1: TỔNG QUAN VỀ GIẢI THUẬT VÀ CẤU TRÚC DỮ LIỆU.....	3
1.1. VAI TRÒ CỦA CẤU TRÚC DỮ LIỆU TRONG MỘT ĐỀ ÁN TIN HỌC.....	3
1.1.1. Mối liên hệ giữa cấu trúc dữ liệu và giải thuật	3
1.1.2. Các tiêu chuẩn đánh giá cấu trúc dữ liệu.....	5
1.2. TRỪU TƯỢNG HOÁ DỮ LIỆU.....	5
1.2.1. Định nghĩa kiểu dữ liệu	6
1.2.2. Các kiểu dữ liệu cơ bản	6
1.2.3. Các kiểu dữ liệu có cấu trúc	7
1.2.4. Một số kiểu dữ liệu có cấu trúc cơ bản.....	8
CHƯƠNG 2: CÁC PHƯƠNG PHÁP TÌM KIẾM VÀ SẮP XẾP CƠ BẢN	12
2.1. CÁC PHƯƠNG PHÁP TÌM KIẾM CƠ BẢN.....	12
2.1.1. Nhu cầu tìm kiếm dữ liệu	12
2.1.2. Các giải thuật tìm kiếm nội.....	12
2.2. CÁC PHƯƠNG PHÁP SẮP XẾP CƠ BẢN.....	16
2.2.1. Định nghĩa bài toán sắp xếp.....	16
2.2.2. Phương pháp chọn trực tiếp (Selection Sort)	16
2.2.3. Phương pháp chèn trực tiếp (Selection Sort).....	18
2.2.4. Phương pháp đổi chỗ trực tiếp (Interchange Sort).....	20
2.2.5. Phương pháp nổi bọt (Bubble Sort).....	23
2.2.6. Phương pháp sắp xếp cây (Heap Sort)	26
2.2.7. Phương pháp sắp xếp phân hoạch (Quick Sort)	30
CHƯƠNG 3: DANH SÁCH LIÊN KẾT	34
3.1. DANH SÁCH LIÊN KẾT (LINK LIST).....	34
3.1.1. Định nghĩa:	34
3.1.2. Các hình thức tổ chức danh sách.....	34
3.2. DANH SÁCH LIÊN KẾT ĐƠN.....	35
3.2.1. Tổ chức danh sách liên kết đơn	35
3.2.2. Các thao tác cơ bản trên danh sách liên kết đơn.....	36
3.3. DANH SÁCH LIÊN KẾT KÉP.....	41
3.3.1. Tổ chức danh sách liên kết kép.....	41
3.3.2. Các thao tác cơ bản trên danh sách kép.....	42
CHƯƠNG 4: CẤU TRÚC DỮ LIỆU NGẮN XẾP (STACK) VÀ HÀNG ĐỢI (QUEUE) ..	52
4.1. NGẮN XẾP (STACK)	52
4.1. 1. Biểu diễn Stack dùng mảng:	52
4.1. 2. Biểu diễn Stack dùng danh sách:	53
4.1. 3. Ứng dụng của Stack.....	53
4.2. HÀNG ĐỢI (QUEUE).....	54
4.2.1. Biểu diễn dùng mảng:	54
4.2.2. Dùng danh sách liên kết:	55
4.2.3. Ứng dụng của hàng đợi:	55
CHƯƠNG 5: CÂY VÀ CÂY NHỊ PHÂN	57
5.1. CẤU TRÚC CÂY	57
5.1.1. Một số khái niệm cơ bản.....	57
5.1.2. Một số ví dụ cấu trúc cây.....	57
5.2. CÂY NHỊ PHÂN	58
5.2.1. Định nghĩa	58
5.2.2. Một số tính chất của cây nhị phân.....	59
5.2.3. Biểu diễn cây nhị phân T	59
5.2.4. Duyệt cây nhị phân	59
5.2.5. Biểu diễn cây tổng quát bằng cây nhị phân.....	61
5.2.6. Một cách biểu diễn cây nhị phân khác.....	62
CHƯƠNG 6: CÂY NHỊ PHÂN TÌM KIẾM.....	64

6.1. CÂY NHỊ PHÂN TÌM KIẾM.....	64
6.2. CÁC THAO TÁC TRÊN CÂY NHỊ PHÂN TÌM KIẾM.....	64
6.2.1. Duyệt cây	64
6.2.2. Tìm một phần tử X trong cây	64
6.2.3. Thêm một phần tử X vào cây.....	65
6.2.4. Hủy một phần tử có khóa X	66
6.2.5. Tạo một cây nhị phân tìm kiếm.....	68
6.2.6. Hủy toàn bộ cây nhị phân tìm kiếm	68
CHƯƠNG 7: ĐÁNH GIÁ ĐỘ PHỨC TẠP THUẬT TOÁN.....	70
7.1. CÁC BƯỚC PHÂN TÍCH THUẬT TOÁN.....	71
7.2. SỰ PHÂN LỚP CÁC THUẬT TOÁN	71
7.2.1. Hằng số:.....	71
7.2.2. Hàm $\log N$:	72
7.2.3. Hàm N :.....	72
7.2.4. Hàm $N \log N$:.....	72
7.2.5. Hàm N^2 :.....	72
7.2.6. Hàm N^3 :.....	72
7.2.7. Hàm $2N$:.....	72
7.3. PHÂN TÍCH TRƯỜNG HỢP TRUNG BÌNH.....	73
7.4. SỰ CẦN THIẾT PHẢI PHÂN TÍCH GIẢI THUẬT.....	73
7.5. THỜI GIAN THỰC HIỆN CỦA CHƯƠNG TRÌNH.....	74
7.5.1. Thời gian thực hiện chương trình.	74
7.5.2. Đơn vị đo thời gian thực hiện.	74
7.5.3. Thời gian thực hiện trong trường hợp xấu nhất.	74
7.6. TỶ SUẤT TĂNG VÀ ĐỘ PHỨC TẠP CỦA GIẢI THUẬT.....	74
7.6.1. Tỷ suất tăng.....	74
7.6.2. Khái niệm độ phức tạp của giải thuật.....	75
7.7. CÁCH TÍNH ĐỘ PHỨC TẠP	75
7.7.1 . Qui tắc cộng.....	75
7.7.2. Qui tắc nhân	75
7.7.3. Qui tắc tổng quát để phân tích một chương trình:	76
7.7.4. Độ phức tạp của chương trình có gọi chương trình con không đệ qui.....	76
7.8. PHÂN TÍCH CÁC CHƯƠNG TRÌNH ĐỆ QUY.....	78
7.8.1. Thành lập phương trình đệ quy	78
7.8.2. Giải phương trình đệ quy.....	80
TÀI LIỆU THAM KHẢO	85

CHƯƠNG 1

TỔNG QUAN VỀ GIẢI THUẬT VÀ CẤU TRÚC DỮ LIỆU

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể:

- Hiểu được vai trò của cấu trúc dữ liệu trong đề án tin học.
- Biết được tiêu chuẩn đánh giá cấu trúc dữ liệu.
- Biết và vận dụng được các kiểu dữ liệu vào thực tế.

1.1. VAI TRÒ CỦA CẤU TRÚC DỮ LIỆU TRONG MỘT ĐỀ ÁN TIN HỌC

1.1.1. *Mối liên hệ giữa cấu trúc dữ liệu và giải thuật*

Thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết trên máy tính. Một bài toán thực tế bất kỳ đều bao gồm các đối tượng dữ liệu và các yêu cầu xử lý trên những đối tượng đó. Vì thế, để xây dựng một mô hình tin học phản ánh được bài toán thực tế cần chú trọng đến hai vấn đề:

Tổ chức biểu diễn các đối tượng thực tế: Các thành phần dữ liệu thực tế đa dạng, phong phú và thường chứa đựng những quan hệ nào đó với nhau, do đó trong mô hình tin học của bài toán, cần phải tổ chức, xây dựng các cấu trúc thích hợp nhất sao cho vừa có thể phản ánh chính xác các dữ liệu thực tế này, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này được gọi là xây dựng cấu trúc dữ liệu cho bài toán.

Xây dựng các thao tác xử lý dữ liệu: Từ những yêu cầu xử lý thực tế, cần tìm ra các giải thuật tương ứng để xác định trình tự các thao tác máy tính phải thi hành để cho ra kết quả mong muốn, đây là bước xây dựng giải thuật cho bài toán.

Tuy nhiên khi giải quyết một bài toán trên máy tính, chúng ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng giải thuật mà quên đi tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý, còn đối tượng xử lý của giải thuật lại là dữ liệu, chính dữ liệu chứa đựng các thông tin cần thiết để thực hiện giải thuật. Để xác định được giải thuật phù hợp cần phải biết nó tác động đến loại dữ liệu nào (ví dụ để làm nhuyễn các hạt đậu, người ta dùng cách xay chứ không băm bằng dao, vì đậu sẽ văng ra ngoài) và khi chọn lựa cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào sẽ tác động đến nó (ví dụ để biểu diễn các điểm số của sinh viên người ta dùng số thực thay vì chuỗi ký tự vì còn phải thực hiện thao tác tính trung bình từ những điểm số đó). Như vậy trong một đề án tin học, giải thuật và cấu trúc dữ liệu có mối quan hệ chặt chẽ với nhau, được thể hiện qua công thức:

Cấu trúc dữ liệu + Giải thuật = Chương trình

Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tương ứng, phù hợp. Khi cấu trúc dữ liệu thay đổi thường giải thuật cũng phải thay đổi theo để tránh việc xử lý gượng ép, thiếu tự nhiên trên một cấu trúc không phù hợp. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp giải thuật xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa đáp ứng nhanh vừa tiết kiệm vật tư, giải thuật cũng dễ hiểu và đơn giản hơn.

Ví dụ: Một chương trình quản lý điểm thi của sinh viên cần lưu trữ các điểm số của 3 sinh viên. Do mỗi sinh viên có 4 điểm số ứng với 4 môn học khác nhau nên dữ liệu có dạng bảng như sau:

Sinh viên	Môn 1	Môn 2	Môn3	Môn4
SV 1	7	9	5	2
SV 2	5	0	9	4
SV 3	6	3	7	4

Chỉ xét thao tác xử lý là xuất điểm số các môn của từng sinh viên.

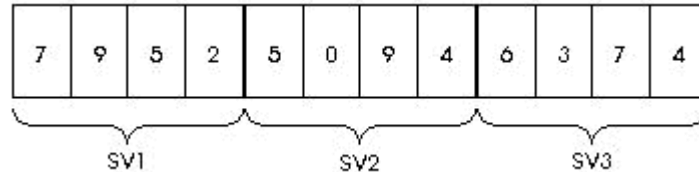
Giả sử có các phương án tổ chức lưu trữ sau:

Phương án 1: Sử dụng mảng một chiều

Có tất cả $3(SV) \times 4(Môn) = 12$ điểm số cần lưu trữ, do đó khai báo mảng result như sau:

```
int result [ 12 ] = { 7, 9, 5, 2,
                     5, 0, 9, 4,
                     6, 3, 7, 4 };
```

Khi đó trong mảng result các phần tử sẽ được lưu trữ như sau:



Và truy xuất điểm số môn j của sinh viên i - là phần tử tại (dòng i, cột j) trong bảng - phải sử dụng một công thức xác định chỉ số tương ứng trong mảng result:

bảngđiểm(dòng i, cột j) \Rightarrow result[((i-1)*số cột) + j]

Ngược lại, với một phần tử bất kỳ trong mảng, muốn biết đó là điểm số của sinh viên nào, môn gì, phải dùng công thức xác định sau:

result[i] \Rightarrow bảngđiểm (dòng((i / số cột) + 1), cột (i % số cột))

Với phương án này, thao tác xử lý được cài đặt như sau:

```
void XuatDiem() //Xuất điểm số của tất cả sinh viên
{
    const int so_mon = 4;
    int sv, mon;
    for (int i=0; i<12; i++)
    {
        sv = i/so_mon; mon = i % so_mon;
        printf("Điểm môn %d của sv %d là: %d", mon, sv,
            result[i]);
    }
}
```

Phương án 2: Sử dụng mảng 2 chiều

Khai báo mảng 2 chiều result có kích thước 3 dòng* 4 cột như sau:

```
int result[3][4] = { {7,9,5,2}, {5,0,9,4}, {6,3,7,4} };
```

Khi đó trong mảng result các phần tử sẽ được lưu trữ như sau:

	Cột 0	Cột 1	Cột 2	Cột 3
Dòng 0	result[0][0] = 7	result[0][1] = 9	result[0][2] = 5	result[0][3] = 2
Dòng 1	result[1][0] = 5	result[1][1] = 0	result[1][2] = 9	result[1][3] = 4
Dòng 2	result[2][0] = 6	result[2][1] = 3	result[2][2] = 7	result[2][3] = 4

Và truy xuất điểm số môn j của sinh viên i - là phần tử tại (dòng i, cột j) trong bảng - cũng chính là phần tử nằm ở vị trí (dòng i, cột j) trong mảng.

bảngđiểm(dòng i, cột j) \Rightarrow result[i][j]

Với phương án này, thao tác xử lý được cài đặt như sau:

```
void XuatDiem() //Xuất điểm số của tất cả sinh viên
{
    int so_mon = 4, so_sv = 3;
    for ( int i=0; i<so_sv; i++)
        for ( int j=0; j<so_mon; j++)
            printf("Điểm môn %d của sv %d là: %d", j, i,
                result[i][j]);
}
```

Nhận xét:

Có thể thấy rõ phương án 2 cung cấp một cấu trúc lưu trữ phù hợp với dữ liệu thực tế hơn phương án 1, và do vậy giải thuật xử lý trên cấu trúc dữ liệu của phương án 2 cũng đơn giản, tự nhiên hơn.

1.1.2. Các tiêu chuẩn đánh giá cấu trúc dữ liệu

Do tầm quan trọng đã được trình bày trong phần 1, nhất thiết phải chú trọng đến việc lựa chọn một phương án tổ chức dữ liệu thích hợp cho đề án. Một cấu trúc dữ liệu tốt phải thỏa mãn các tiêu chuẩn sau:

Phản ánh đúng thực tế : Đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình sống để có thể chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế.

Ví dụ: Một số tình huống chọn cấu trúc lưu trữ sai

- Chọn một biến số nguyên int để lưu trữ tiền thưởng bán hàng (được tính theo công thức tiền thưởng bán hàng = trị giá hàng * 5%), do vậy sẽ làm tròn mọi giá trị tiền thưởng gây thiệt hại cho nhân viên bán hàng. Trường hợp này phải sử dụng biến số thực để phản ánh đúng kết quả của công thức tính thực tế.

- Trong trường trung học, mỗi lớp có thể nhận tối đa 28 học sinh. Lớp hiện có 20 học sinh, mỗi tháng mỗi học sinh đóng học phí \$10. Chọn một biến số nguyên unsigned char (khả năng lưu trữ 0 - 255) để lưu trữ tổng học phí của lớp học trong tháng, nếu xảy ra trường hợp có thêm 6 học sinh được nhận vào lớp thì giá trị tổng học phí thu được là \$260, vượt khỏi khả năng lưu trữ của biến đã chọn, gây ra tình trạng tràn, sai lệch.

Phù hợp với các thao tác trên đó: Tiêu chuẩn này giúp tăng tính hiệu quả của đề án: việc phát triển các thuật toán đơn giản, tự nhiên hơn; chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

Ví dụ: Một tình huống chọn cấu trúc lưu trữ không phù hợp

Cần xây dựng một chương trình soạn thảo văn bản, các thao tác xử lý thường xảy ra là chèn, xóa, sửa các ký tự trên văn bản. Trong thời gian xử lý văn bản, nếu chọn cấu trúc lưu trữ văn bản trực tiếp lên tập tin thì sẽ gây khó khăn khi xây dựng các giải thuật cập nhật văn bản và làm chậm tốc độ xử lý của chương trình vì phải làm việc trên bộ nhớ ngoài. Trường hợp này nên tìm một cấu trúc dữ liệu có thể tổ chức ở bộ nhớ trong để lưu trữ văn bản suốt thời gian soạn thảo.

Lưu ý:

Đối với mỗi ứng dụng, cần chú ý đến thao tác nào được sử dụng nhiều nhất để lựa chọn cấu trúc dữ liệu cho thích hợp.

Tiết kiệm tài nguyên hệ thống: Cấu trúc dữ liệu chỉ nên sử dụng tài nguyên hệ thống vừa đủ để đảm nhiệm được chức năng của nó. Thông thường có 2 loại tài nguyên cần lưu tâm nhất: CPU và bộ nhớ. Tiêu chuẩn này nên cân nhắc tùy vào tình huống cụ thể khi thực hiện đề án. Nếu tổ chức sử dụng đề án cần có những xử lý nhanh thì khi chọn cấu trúc dữ liệu yếu tố tiết kiệm thời gian xử lý phải đặt nặng hơn tiêu chuẩn sử dụng tối ưu bộ nhớ, và ngược lại.

Ví dụ: Một số tình huống chọn cấu trúc lưu trữ lãng phí:

- Sử dụng biến int (2 bytes) để lưu trữ một giá trị cho biết tháng hiện hành. Biết rằng tháng chỉ có thể nhận các giá trị từ 1-12, nên chỉ cần sử dụng kiểu char (1 byte) là đủ.

- Để lưu trữ danh sách học viên trong một lớp, sử dụng mảng 50 phần tử (giới hạn số học viên trong lớp tối đa là 50). Nếu số lượng học viên thật sự ít hơn 50, thì gây lãng phí. Trường hợp này cần có một cấu trúc dữ liệu linh động hơn mảng - ví dụ danh sách liên kết - sẽ được bàn đến trong các chương sau.

1.2. TRỪU TƯỢNG HOÁ DỮ LIỆU

Máy tính thực sự chỉ có thể lưu trữ dữ liệu ở dạng nhị phân thô sơ. Nếu muốn phản ánh

được dữ liệu thực tế đa dạng và phong phú, cần phải xây dựng những phép ánh xạ, những qui tắc tổ chức phức tạp che lên tầng dữ liệu thô, nhằm đưa ra những khái niệm logic về hình thức lưu trữ khác nhau thường được gọi là kiểu dữ liệu. Như đã phân tích ở phần 1, giữa hình thức lưu trữ dữ liệu và các thao tác xử lý trên đó có quan hệ mật thiết với nhau. Từ đó có thể đưa ra một định nghĩa cho kiểu dữ liệu như sau:

1.2.1. Định nghĩa kiểu dữ liệu

Kiểu dữ liệu T được xác định bởi một bộ $\langle V, O \rangle$, với:

V : tập các giá trị hợp lệ mà một đối tượng kiểu T có thể lưu trữ.

O : tập các thao tác xử lý có thể thi hành trên đối tượng kiểu T.

Ví dụ:

Giả sử có kiểu dữ liệu mẫu tự = $\langle V_c, O_c \rangle$ với

$$V_c = \{ a-z, A-Z \}$$

$O_c = \{ \text{lấy mã ASCII của ký tự, biến đổi ký tự thường thành ký tự hoa...} \}$

Giả sử có kiểu dữ liệu số nguyên = $\langle V_i, O_i \rangle$ với

$$V_i = \{ -32768..32767 \}$$

$$O_i = \{ +, -, *, /, \% \}$$

Như vậy, muốn sử dụng một kiểu dữ liệu cần nắm vững cả nội dung dữ liệu được phép lưu trữ và các xử lý tác động trên đó.

Các thuộc tính của 1 KDL bao gồm:

- Tên KDL
- Miền giá trị
- Kích thước lưu trữ
- Tập các toán tử tác động lên KDL

1.2.2. Các kiểu dữ liệu cơ bản

Các loại dữ liệu cơ bản thường là các loại dữ liệu đơn giản, không có cấu trúc. Chúng thường là các giá trị vô hướng như các số nguyên, số thực, các ký tự, các giá trị logic ... Các loại dữ liệu này, do tính thông dụng và đơn giản của mình, thường được các ngôn ngữ lập trình (NNLT) cấp cao xây dựng sẵn như một thành phần của ngôn ngữ để giảm nhẹ công việc cho người lập trình. Chính vì vậy đôi khi người ta còn gọi chúng là các kiểu dữ liệu định sẵn.

Thông thường, các kiểu dữ liệu cơ bản bao gồm:

Kiểu có thứ tự rời rạc: số nguyên, ký tự, logic, liệt kê, miền con.

Kiểu không rời rạc: số thực.

Tùy ngôn ngữ lập trình, các kiểu dữ liệu định nghĩa sẵn có thể khác nhau đôi chút. Với ngôn ngữ C, các kiểu dữ liệu này chỉ gồm số nguyên, số thực, ký tự. Và theo quan điểm của C, kiểu ký tự thực chất cũng là kiểu số nguyên về mặt lưu trữ, chỉ khác về cách sử dụng. Ngoài ra, giá trị logic ĐÚNG (TRUE) và giá trị logic SAI (FALSE) được biểu diễn trong C như là các giá trị nguyên khác zero và zero. Trong khi đó PASCAL định nghĩa tất cả các kiểu dữ liệu cơ sở đã liệt kê ở trên và phân biệt chúng một cách chặt chẽ. Trong giới hạn giáo trình này ngôn ngữ chính dùng để minh họa sẽ là C.

Các kiểu dữ liệu định sẵn trong C gồm các kiểu sau:

Tên kiểu	Kích thước	Miền giá trị	Ghi chú
Char	01 byte	-128 đến 127	Có thể dùng như số nguyên 1 byte có dấu hoặc kiểu ký tự
unsign char	01 byte	0 đến 255	Số nguyên 1 byte không dấu
int	02 byte	-32768 đến 32767	
unsign int	02 byte	0 đến 65535	Có thể gọi tắt là unsign
long	04 byte	-2^{32} đến $2^{31}-1$	
unsign long	04 byte	0 đến $2^{32}-1$	

Tên kiểu	Kthước	Miền giá trị	Ghi chú
float	04 byte	3.4E-38 đến 3.4E38	Giới hạn chỉ trị tuyệt đối. Các giá trị <3.4E-38 được coi = 0. Tuy nhiên kiểu float chỉ có 7 chữ số có nghĩa.
double	08 byte	1.7E-308 đến 1.7E308	
long double	10 byte	3.4E-4932 đến 1.1E4932	

Một số điều đáng lưu ý đối với các kiểu dữ liệu cơ bản trong C là kiểu ký tự (char) có thể dùng theo hai cách (số nguyên 1 byte hoặc ký tự). Ngoài ra C không định nghĩa kiểu logic (boolean) mà nó đơn giản đồng nhất một giá trị nguyên khác 0 với giá trị TRUE và giá trị 0 với giá trị FALSE khi có nhu cầu xét các giá trị logic. Như vậy, trong C xét cho cùng chỉ có 2 loại dữ liệu cơ bản là số nguyên và số thực. Tức là chỉ có dữ liệu số. Hơn nữa các số nguyên trong C có thể được thể hiện trong 3 hệ cơ số là hệ thập phân, hệ thập lục phân và hệ bát phân. Nhờ những quan điểm trên, C rất được những người lập trình chuyên nghiệp thích dùng.

Các kiểu cơ sở rất đơn giản và không thể hiện rõ sự tổ chức dữ liệu trong một cấu trúc, thường chỉ được sử dụng làm nền để xây dựng các kiểu dữ liệu phức tạp khác.

1.2.3. Các kiểu dữ liệu có cấu trúc

Tuy nhiên trong nhiều trường hợp, chỉ với các kiểu dữ liệu cơ sở không đủ để phản ánh tự nhiên và đầy đủ bản chất của sự vật thực tế, dẫn đến nhu cầu phải xây dựng các kiểu dữ liệu mới dựa trên việc tổ chức, liên kết các thành phần dữ liệu có kiểu dữ liệu đã được định nghĩa. Những kiểu dữ liệu được xây dựng như thế gọi là kiểu dữ liệu có cấu trúc. Đa số các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như mảng, chuỗi, tập tin, bản ghi...và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

Ví dụ: Để mô tả một đối tượng sinh viên, cần quan tâm đến các thông tin sau:

Mã sinh viên: chuỗi ký tự
 Tên sinh viên: chuỗi ký tự
 Ngày sinh: kiểu ngày tháng
 Nơi sinh: chuỗi ký tự
 Điểm thi: số thực

Các kiểu dữ liệu cơ sở cho phép mô tả một số thông tin như:

float Diemthi;

Các thông tin khác đòi hỏi phải sử dụng các kiểu có cấu trúc như:

```
char masv[15];
char tensv[15];
char noisinh[15];
```

Để thể hiện thông tin về ngày tháng năm sinh cần phải xây dựng một kiểu bản ghi,

```
typedef struct tagDate{
    char ngay;
    char thang;
    char thang;
```

```
}Date;
```

Cuối cùng, ta có thể xây dựng kiểu dữ liệu thể hiện thông tin về một sinh viên:

```
typedef struct tagSinhVien
{
    char masv[15];
    char tensv[15];
    Date ngaysinh;
    char noisinh[15];
    int Diem thi;
}SinhVien;
```

Giả sử đã có cấu trúc phù hợp để lưu trữ một sinh viên, nhưng thực tế lại cần quản lý

nhiều sinh viên, lúc đó nảy sinh nhu cầu xây dựng kiểu dữ liệu mới...Mục tiêu của việc nghiên cứu cấu trúc dữ liệu chính là tìm những phương cách thích hợp để tổ chức, liên kết dữ liệu, hình thành các kiểu dữ liệu có cấu trúc từ những kiểu dữ liệu đã được định nghĩa.

1.2.4. Một số kiểu dữ liệu có cấu trúc cơ bản

a. Kiểu chuỗi ký tự

Chuỗi ký tự là một trong các kiểu dữ liệu có cấu trúc đơn giản nhất và thường các ngôn ngữ lập trình đều định nghĩa nó như một kiểu cơ bản. Do tính thông dụng của kiểu chuỗi ký tự các ngôn ngữ lập trình luôn cung cấp sẵn một bộ các hàm thư viện các xử lý trên kiểu dữ liệu này. Đặc biệt trong C thư viện các hàm xử lý chuỗi ký tự rất đa dạng và phong phú. Các hàm này được đặt trong thư viện string.lib của C.

Chuỗi ký tự trong C được cấu trúc như một chuỗi liên tiếp các ký tự kết thúc bằng ký tự có mã ASCII bằng 0 (NULL character). Như vậy, giới hạn chiều dài của một chuỗi ký tự trong C là 1 Segment (tối đa chứa 65535 ký tự), ký tự đầu tiên được đánh số là ký tự thứ 0.

Ta có thể khai báo một chuỗi ký tự theo một số cách sau đây:

```
char S[10]; //Khai báo một chuỗi ký tự S có chiều dài
           // tối đa 10 (kể cả kí tự kết thúc)
char S[]="ABC"; // Khai báo một chuỗi ký tự S có chiều
               // dài bằng chiều dài của chuỗi "ABC"
               // và giá trị khởi đầu của S là "ABC"
```

```
char *S="ABC"; //Giống cách khai báo trên.
```

Trong ví dụ trên ta cũng thấy được một hằng chuỗi ký tự được thể hiện bằng một chuỗi ký tự đặt trong cặp ngoặc kép "".

Các thao tác trên chuỗi ký tự rất đa dạng. Sau đây là một số thao tác thông dụng:

Thao tác	Hàm trong C
So sánh 2 chuỗi	strcmp
Sao chép 2 chuỗi	strcpy
Kiểm tra 1 chuỗi nằm trong chuỗi kia	strstr
Cắt 1 từ ra khỏi 1 chuỗi	strtok
Đổi 1 số ra chuỗi	itoa
Đổi 1 chuỗi ra số	Atoi atof...
Đổi 1 hay 1 số giá trị ra chuỗi	sprintf
Nhập một chuỗi	gets
Xuất một chuỗi	puts

b. Kiểu mảng

Kiểu dữ liệu mảng là kiểu dữ liệu trong đó mỗi phần tử của nó là một tập hợp có thứ tự các giá trị có cùng cấu trúc được lưu trữ liên tiếp nhau trong bộ nhớ. Mảng có thể một chiều hay nhiều chiều. Một dãy số chính là hình tượng của mảng 1 chiều, ma trận là hình tượng của mảng 2 chiều.

Một điều đáng lưu ý là mảng 2 chiều có thể coi là mảng một chiều trong đó mỗi phần tử của nó là 1 mảng một chiều. Tương tự như vậy, một mảng n chiều có thể coi là mảng 1 chiều trong đó mỗi phần tử là 1 mảng n-1 chiều.

Hình tượng này được thể hiện rất rõ trong cách khai báo của C.

Mảng 1 chiều được khai báo như sau:

<Kiểu dữ liệu> <Tên biến>[<Số phần tử>];

Ví dụ để khai báo một biến có tên a là một mảng nguyên 1 chiều có tối đa 100 phần tử ta

phải khai báo như sau:

```
int a[100];
```

Ta cũng có thể vừa khai báo vừa gán giá trị khởi động cho một mảng như sau:

```
int a[5] = (1, 7, -3, 8, 19);
```

Trong trường hợp này C cho phép ta khai báo một cách tiện lợi hơn

```
int a[] = (1, 7, -3, 8, 19);
```

Như ta thấy, ta không cần chỉ ra số lượng phần tử cụ thể trong khai báo. Trình biên dịch của C sẽ tự động làm việc này cho chúng ta.

Tương tự ta có thể khai báo một mảng 2 chiều hay nhiều chiều theo cú pháp sau:

```
<Kiểu dữ liệu> <Tên biến>[<Số phần tử1>][<Số phần tử2>]...;
```

Ví dụ, ta có thể khai báo:

```
int a[100][150];
```

hay

```
int a[][] = {{1, 7, -3, 8, 19}, {4, 5, 2, 8, 9}, {21, -7, 45, -3, 4}}; mảng a sẽ có kích thước là 3x5).
```

c. Kiểu mẫu tin (cấu trúc)

Nếu kiểu dữ liệu mảng là kiểu dữ liệu trong đó mỗi phần tử của nó là một tập hợp có thứ tự các giá trị có cùng cấu trúc được lưu trữ liên tiếp nhau trong bộ nhớ thì mẫu tin là kiểu dữ liệu mà trong đó mỗi phần tử của nó là tập hợp các giá trị có thể khác cấu trúc. Kiểu mẫu tin cho phép chúng ta mô tả các đối tượng có cấu trúc phức tạp.

Khai báo tổng quát của kiểu struct như sau:

```
typedef struct <tên kiểu struct>
```

```
{
```

```
    <KDL> <tên trường>;
```

```
    <KDL> <tên trường>;
```

```
    ...
```

```
}[<Name>];
```

Ví dụ để mô tả các thông tin về một con người ta có thể khai báo một kiểu dữ liệu như sau:

```
struct tagNguoi
```

```
{
```

```
    char HoTen[35];
```

```
    int NamSinh;
```

```
    char NoiSinh[40];
```

```
    char GioiTinh; //0: Nữ, 1: Nam
```

```
    char DiaChi[50];
```

```
    char Ttgd; //0: Không có gia đình, 1: Có gia đình
```

```
} Nguoi;
```

Kiểu mẫu tin bổ sung những thiếu sót của kiểu mảng, giúp ta có khả năng thể hiện các đối tượng đa dạng của thế giới thực vào trong máy tính một cách dễ dàng và chính xác hơn.

d. Kiểu union

Kiểu union là một dạng cấu trúc dữ liệu đặc biệt của ngôn ngữ C. Nó rất giống với kiểu struct. Chỉ khác một điều, trong kiểu union, các trường được phép dùng chung một vùng nhớ. Hay nói cách khác, cùng một vùng nhớ ta có thể truy xuất dưới các dạng khác nhau.

Khai báo tổng quát của kiểu union như sau:

```
typedef union <tên kiểu union>{
```

```
    <KDL> <tên trường>;
```

```
    <KDL> <tên trường>;
```

```
    ...
```

```
}[<Name>];
```

Ví dụ, ta có thể định nghĩa kiểu số sau:

```
typedef union tagNumber{
    int i;
    long l;
}Number;
```

Việc truy xuất đến một trường trong union được thực hiện hoàn toàn giống như trong struct. Giả sử có biến n kiểu Number. Khi đó, n.i cho ta một số kiểu int còn n.l cho ta một số kiểu long, nhưng cả hai đều dùng chung một vùng nhớ. Vì vậy, khi ta gán n.l = 0xfd03; thì giá trị của n.i cũng bị thay đổi (n.i sẽ bằng 3);

Việc dùng kiểu union rất có lợi khi cần khai báo các CTDL mà nội dung của nó thay đổi tùy trạng thái. Ví dụ để mô tả các thông tin về một con người ta có thể khai báo một kiểu dữ liệu như sau:

```
struct tagNguoi
{
    char HoTen[35];
    int    NamSinh;
    char NoiSinh[40];
    char GioiTinh;    //0: Nữ, 1: Nam
    char DiaChi[50];
    char Ttgd;        //0:Không có gia đình, 1: Có gia đình
    union {
        char tenVo[35];
        char tenChong[35];
    }
}Nguoi;
```

Tùy theo người mà ta đang xét là nam hay nữ ta sẽ truy xuất thông tin qua trường có tên tenVo hay tenChong.

❖ Bài tập củng cố:

- Viết chương trình quản lý một tập tin văn bản theo các yêu cầu:
 - Nhập từ bàn phím nội dung một văn bản sau đó ghi vào đĩa.
 - Đọc từ đĩa nội dung văn bản vừa nhập và in lên màn hình.
 - Đọc từ đĩa nội dung văn bản vừa nhập, in nội dung đó lên màn hình và cho phép nối thêm thông tin vào cuối tập tin đó.
- Viết chương trình cho phép thống kê số lần xuất hiện của các ký tự là chữ ('A'..'Z', 'a'..'z') trong một tập tin văn bản.
- Viết chương trình đếm số từ và số dòng trong một tập tin văn bản.
- Viết chương trình nhập từ bàn phím và ghi vào 1 tập tin tên là DMHH.DAT với mỗi phần tử của tập tin là 1 cấu trúc bao gồm các trường: Ma (mã hàng: char[5]), Ten (Tên hàng: char[20]). Kết thúc việc nhập bằng cách nhập Ma bằng '0'. Ta sẽ dùng tập tin này để giải mã hàng hóa cho tập tin DSHH.DAT sẽ đề cập trong bài 5.
- Viết chương trình cho phép nhập từ bàn phím và ghi vào 1 tập tin tên DSHH.Dat với mỗi phần tử của tập tin là một cấu trúc bao gồm các trường: mh (mã hàng: char[5]), sl (số lượng : int), dg (đơn giá: float), st (Số tiền: float) theo yêu cầu:
 - Mỗi lần nhập một cấu trúc
 - Trước tiên nhập mã hàng (mh), đưa mh so sánh với ma trong tập tin DMHH.DAT đã được tạo ra bởi bài tập 1, nếu mh=ma thì in tên hàng ngay bên cạnh mã hàng.
 - Nhập số lượng (sl).
 - Nhập đơn giá (dg).
 - Tính số tiền = số lượng * đơn giá.

STT	MA HANG	TEN HANG	SO LG	DON GIA	SO TIEN
1	A0101	Sua Co gai Ha Lan	10	20000	200000
2	B0101	Sua Ong Tho	15	10000	150000

CHƯƠNG 2

CÁC PHƯƠNG PHÁP TÌM KIẾM VÀ SẮP XẾP CƠ BẢN

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể:

- Vận dụng, cài đặt và đánh giá được phương pháp tìm kiếm tuyến tính và nhị phân.
- Vận dụng, cài đặt và đánh giá được các phương pháp sắp xếp: Selection Sort, Insertion Sort, Bubble Sort, Interchange Sort, Quick Sort và Heap Sort

2.1. CÁC PHƯƠNG PHÁP TÌM KIẾM CƠ BẢN

2.1.1. Nhu cầu tìm kiếm dữ liệu

Trong hầu hết các hệ lưu trữ, quản lý dữ liệu, thao tác tìm kiếm thường được sử dụng thường xuyên để khai thác thông tin:

Ví dụ: tra cứu từ điển, tìm sách trong thư viện...

Do các hệ thống thông tin thường phải lưu trữ một khối lượng dữ liệu đáng kể, nên việc xây dựng các giải thuật cho phép tìm kiếm nhanh sẽ có ý nghĩa rất lớn. Nếu dữ liệu trong hệ thống đã được tổ chức theo một trật tự nào đó, thì việc tìm kiếm sẽ tiến hành nhanh chóng và hiệu quả hơn:

Ví dụ: các từ trong từ điển được sắp xếp theo từng vần, trong mỗi vần lại được sắp xếp theo trình tự alphabet; sách trong thư viện được xếp theo chủ đề ...

Vì thế, khi xây dựng một hệ quản lý thông tin trên máy tính, bên cạnh các thuật toán tìm kiếm, các thuật toán sắp xếp dữ liệu cũng là một trong những chủ đề được quan tâm hàng đầu.

Hiện nay đã có nhiều giải thuật tìm kiếm và sắp xếp được xây dựng, mức độ hiệu quả của từng giải thuật còn phụ thuộc vào tính chất của cấu trúc dữ liệu cụ thể mà nó tác động đến. Dữ liệu được lưu trữ chủ yếu trong bộ nhớ chính và trên bộ nhớ phụ, do đặc điểm khác nhau của thiết bị lưu trữ, các thuật toán tìm kiếm và sắp xếp được xây dựng cho các cấu trúc lưu trữ trên bộ nhớ chính hoặc phụ cũng có những đặc thù khác nhau. Chương này sẽ trình bày các thuật toán sắp xếp và tìm kiếm dữ liệu được lưu trữ trên bộ nhớ chính - gọi là các giải thuật *tìm kiếm và sắp xếp nội*.

2.1.2. Các giải thuật tìm kiếm nội

Có 2 giải thuật thường được áp dụng để tìm kiếm dữ liệu là tìm tuyến tính và tìm nhị phân. Để đơn giản trong việc trình bày giải thuật, bài toán được đặc tả như sau:

Tập dữ liệu được lưu trữ là dãy số a_1, a_2, \dots, a_N .

Giả sử chọn cấu trúc dữ liệu mảng để lưu trữ dãy số này trong bộ nhớ chính, có khai báo:

```
int a[N];
```

Lưu ý các bản cài đặt trong giáo trình sử dụng ngôn ngữ C, do đó chỉ số của mảng mặc định bắt đầu từ 0, nên các giá trị của các chỉ số có chênh lệch so với thuật toán, nhưng ý nghĩa không đổi

Khoá cần tìm là x , được khai báo như sau:

```
int x;
```

2.1.2.1. Tìm kiếm tuyến tính

Giải thuật

Tìm tuyến tính là một kỹ thuật tìm kiếm rất đơn giản và cổ điển. Thuật toán tiến hành so sánh x lần lượt với phần tử thứ nhất, thứ hai, ... của mảng a cho đến khi gặp được phần tử có khóa cần tìm, hoặc đã tìm hết mảng mà không thấy x . Các bước tiến hành như sau:

Bước 1:

$i = 1$; // bắt đầu từ phần tử đầu tiên của dãy

Bước 2: So sánh $a[i]$ với x , có 2 khả năng :

$a[i] = x$: Tìm thấy. Dừng

$a[i] \neq x$: Sang Bước 3.

Bước 3 :

$i = i+1$; // xét tiếp phần tử kế trong mảng

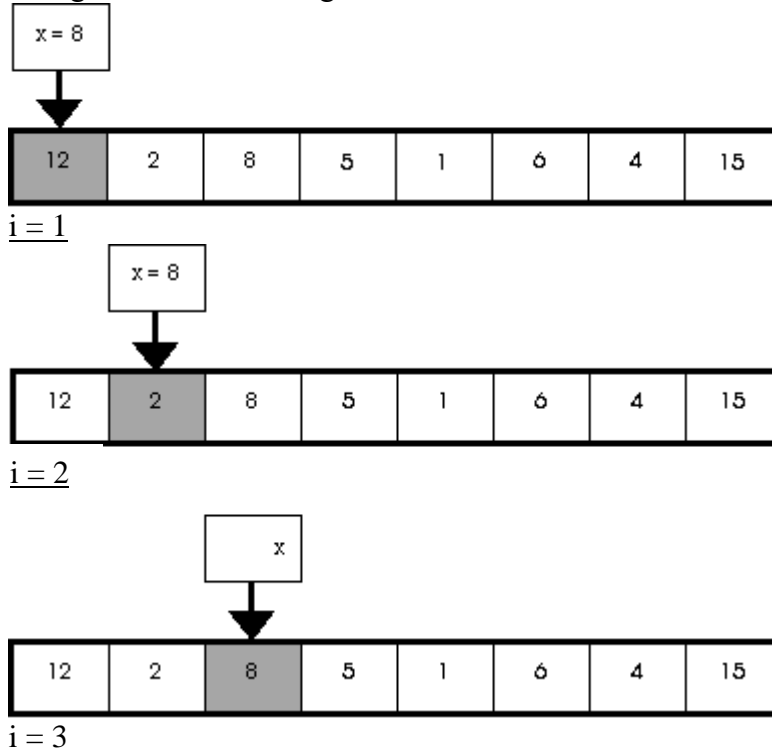
Nếu $i > N$: Hết mảng, không tìm thấy. Dừng

Ngược lại: Lặp lại Bước 2.

Ví dụ:

Cho dãy số a: 12 2 8 5 1 6 4 15

Nếu giá trị cần tìm là 8, giải thuật được tiến hành như sau :



Dừng.

Cài đặt

Từ mô tả trên đây của thuật toán tìm tuyến tính, có thể cài đặt hàm LinearSearch để xác định vị trí của phần tử có khóa x trong mảng a:

```
int LinearSearch(int a[], int N, int x)
{
    int i=0;
    while ((i<N) && (a[i]!=x )) i++;
    if(i==N) return -1; // tìm hết mảng nhưng không có x
    else return i; // a[i] là phần tử có khoá x
}
```

Trong cài đặt trên đây, nhận thấy mỗi lần lặp của vòng lặp while phải tiến thành kiểm tra 2 điều kiện ($i < N$) (điều kiện biên của mảng) và ($a[i] \neq x$) (điều kiện kiểm tra chính). Nhưng thật sự chỉ cần kiểm tra điều kiện chính ($a[i] \neq x$), để cải tiến cài đặt, có thể dùng phương pháp "lính canh" - đặt thêm một phần tử có giá trị x vào cuối mảng, như vậy bảo đảm luôn tìm thấy x trong mảng, sau đó dựa vào vị trí tìm thấy để kết luận. Cài đặt cải tiến sau đây của hàm LinearSearch giúp giảm bớt một phép so sánh trong vòng lặp:

```
int LinearSearch(int a[], int N, int x)
{
    int i=0; // mảng gồm N phần tử từ a[0]..a[N-1]
    a[N] = x; // thêm phần tử thứ N+1
    while (a[i]!=x )
```

```

i++;
if (i==N)
    return -1;        // tìm hết mảng nhưng không có x
else
    return i;         // tìm thấy x tại vị trí i

```

Đánh giá giải thuật

Có thể ước lượng độ phức tạp của giải thuật tìm kiếm qua số lượng các phép so sánh được tiến hành để tìm ra x. Trường hợp giải thuật tìm tuyến tính, có:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử đầu tiên có giá trị x
Xấu nhất	n+1	Phần tử cuối cùng có giá trị x
Trung bình	(n+1)/2	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau.

Vậy giải thuật tìm tuyến tính có độ phức tạp tính toán cấp n: $T(n) = O(n)$

Nhận xét:

Giải thuật tìm tuyến tính không phụ thuộc vào thứ tự của các phần tử mảng, do vậy đây là phương pháp tổng quát nhất để tìm kiếm trên một dãy số bất kỳ.

Một thuật toán có thể được cài đặt theo nhiều cách khác nhau, kỹ thuật cài đặt ảnh hưởng đến tốc độ thực hiện của thuật toán.

2.1.2.2. Tìm kiếm nhị phân

Giải thuật

Đối với những dãy số đã có thứ tự (giả sử thứ tự tăng), các phần tử trong dãy có quan hệ $a_{i-1} \leq a_i \leq a_{i+1}$, từ đó kết luận được nếu $x > a_i$ thì x chỉ có thể xuất hiện trong đoạn $[a_{i+1}, a_n]$ của dãy, ngược lại nếu $x < a_i$ thì x chỉ có thể xuất hiện trong đoạn $[a_1, a_{i-1}]$ của dãy. Giải thuật tìm nhị phân áp dụng nhận xét trên đây để tìm cách giới hạn phạm vi tìm kiếm sau mỗi lần so sánh x với một phần tử trong dãy. Ý tưởng của giải thuật là tại mỗi bước tiến hành so sánh x với phần tử nằm ở vị trí giữa của dãy tìm kiếm hiện hành, dựa vào kết quả so sánh này để quyết định giới hạn dãy tìm kiếm ở bước kế tiếp là nửa trên hay nửa dưới của dãy tìm kiếm hiện hành. Giả sử dãy tìm kiếm hiện hành bao gồm các phần tử $a_{left} .. a_{right}$, các bước tiến hành như sau:

Bước 1:

left = 1; right = N; // tìm kiếm trên tất cả các phần tử

Bước 2:

mid = (left+right)/2; // lấy mốc so sánh

So sánh $a[mid]$ với x, có 3 khả năng :

$a[mid] = x$: Tìm thấy. Dừng

$a[mid] > x$: //tìm tiếp x trong dãy con $a_{left} .. a_{mid-1}$:

right =midle - 1;

$a[mid] < x$: //tìm tiếp x trong dãy con $a_{mid+1} .. a_{right}$:

left = mid+ 1;

Bước 3:

Nếu left > right //còn phần tử chưa xét, tìm tiếp.

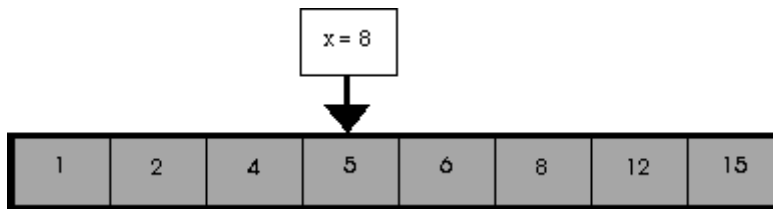
Lặp lại Bước 2.

Ngược lại: Dừng; //Đã xét hết tất cả các phần tử.

Ví dụ:

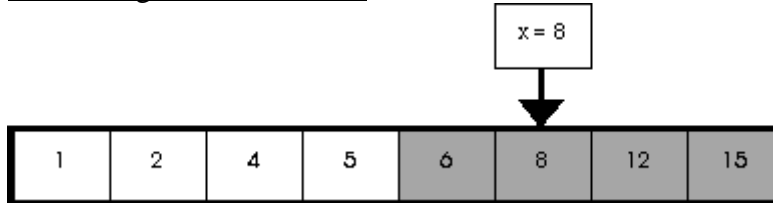
Cho dãy số a gồm 8 phần tử:

1 2 4 5 6 8 12 15



Nếu giá trị cần tìm là 8, giải thuật được tiến hành như sau:

left = 1, right = 8, middle = 4



left = 5, right = 8, middle = 6

Dừng.

Cài đặt

Thuật toán tìm nhị phân có thể được cài đặt thành hàm BinarySearch:

```
int BinarySearch(int a[], int N, int x)
{
    int left = 0; right = N-1;
    int middle;
    do {
        mid = (left + right)/2;
        if (x == a[mid])
            return mid; // Thấy x tại mid
        else
            if (x < a[mid])
                right = mid - 1;
            else left = mid + 1;
    } while (left <= right);
    return -1; // Tìm hết dãy mà không có x
}
```

Đánh giá giải thuật

Trường hợp giải thuật tìm nhị phân, có bảng phân tích sau:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử giữa của mảng có giá trị x
Xấu nhất	$\log_2 n$	Không có x trong mảng
Trung bình	$\log_2 n/2$	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau

Vậy giải thuật tìm nhị phân có độ phức tạp tính toán cấp n: $T(n) = O(\log_2 n)$

Nhận xét:

Giải thuật tìm nhị phân dựa vào quan hệ giá trị của các phần tử mảng để định hướng trong quá trình tìm kiếm, do vậy chỉ áp dụng được cho những dãy đã có thứ tự.

Giải thuật tìm nhị phân tiết kiệm thời gian hơn rất nhiều so với giải thuật tìm tuyến tính do $T_{\text{nhị phân}}(n) = O(\log_2 n) < T_{\text{tuyến tính}}(n) = O(n)$.

Tuy nhiên khi muốn áp dụng giải thuật tìm nhị phân cần phải xét đến thời gian sắp xếp dãy số để thỏa điều kiện dãy số có thứ tự. Thời gian này không nhỏ, và khi dãy số biến động cần phải tiến hành sắp xếp lại. Tất cả các nhu cầu đó tạo ra khuyết điểm chính cho giải thuật tìm nhị phân. Ta cần cân nhắc nhu cầu thực tế để chọn một trong hai giải thuật tìm kiếm trên sao cho

có lợi nhất.

2.2. CÁC PHƯƠNG PHÁP SẮP XẾP CƠ BẢN

2.2.1. Định nghĩa bài toán sắp xếp

Sắp xếp là quá trình xử lý một danh sách các phần tử (hoặc các mẫu tin) để đặt chúng theo một thứ tự thỏa mãn một tiêu chuẩn nào đó dựa trên nội dung thông tin lưu giữ tại mỗi phần tử.

Tại sao cần phải sắp xếp các phần tử thay vì để nó ở dạng tự nhiên (chưa có thứ tự) vốn có? Ví dụ của bài toán tìm kiếm với phương pháp tìm kiếm nhị phân và tuần tự đủ để trả lời câu hỏi này.

Khi khảo sát bài toán sắp xếp, ta sẽ phải làm việc nhiều với một khái niệm gọi là nghịch thế.

Khái niệm nghịch thế:

Xét một mảng các số a_0, a_1, \dots, a_n .

Nếu có $i < j$ và $a_i > a_j$, thì ta gọi đó là một nghịch thế.

Mảng chưa sắp xếp sẽ có nghịch thế.

Mảng đã có thứ tự sẽ không chứa nghịch thế. Khi đó a_0 sẽ là phần tử nhỏ nhất rồi đến a_1, a_2, \dots, a_n . Như vậy, để sắp xếp một mảng, ta có thể tìm cách giảm số các nghịch thế trong mảng này bằng cách hoán vị các cặp phần tử a_i, a_j nếu có $i < j$ và $a_i > a_j$ theo một qui luật nào đó.

Cho trước một dãy số a_1, a_2, \dots, a_N được lưu trữ trong cấu trúc dữ liệu mảng

$\text{int } a[N];$

Sắp xếp dãy số a_1, a_2, \dots, a_N là thực hiện việc bố trí lại các phần tử sao cho hình thành được dãy mới $a_{k1}, a_{k2}, \dots, a_{kN}$ có thứ tự (giả sử xét thứ tự tăng) nghĩa là $a_{ki} < a_{ki+1}$. Mà để quyết định được những tình huống cần thay đổi vị trí các phần tử trong dãy, cần dựa vào kết quả của một loạt phép so sánh. Chính vì vậy, hai thao tác so sánh và gán là các thao tác cơ bản của hầu hết các thuật toán sắp xếp.

Khi xây dựng một thuật toán sắp xếp cần chú ý tìm cách giảm thiểu những phép so sánh và đổi chỗ không cần thiết để tăng hiệu quả của thuật toán. Đối với các dãy số được lưu trữ trong bộ nhớ chính, nhu cầu tiết kiệm bộ nhớ được đặt nặng, do vậy những thuật toán sắp xếp đòi hỏi cấp phát thêm vùng nhớ để lưu trữ dãy kết quả ngoài vùng nhớ lưu trữ dãy số ban đầu thường ít được quan tâm. Thay vào đó, các thuật toán sắp xếp trực tiếp trên dãy số ban đầu (gọi là các thuật toán sắp xếp tại chỗ) lại được đầu tư phát triển. Phần này giới thiệu một số giải thuật sắp xếp từ đơn giản đến phức tạp có thể áp dụng thích hợp cho việc sắp xếp nội.

2.2.2. Phương pháp chọn trực tiếp (Selection Sort)

Giải thuật

Ta thấy rằng, nếu mảng có thứ tự, phần tử a_i luôn là $\min(a_i, a_{i+1}, \dots, a_{n-1})$. Ý tưởng của thuật toán chọn trực tiếp mô phỏng một trong những cách sắp xếp tự nhiên nhất trong thực tế: chọn phần tử nhỏ nhất trong N phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu dãy hiện hành; sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn $N-1$ phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2; lặp lại quá trình trên cho dãy hiện hành... đến khi dãy hiện hành chỉ còn 1 phần tử. Dãy ban đầu có N phần tử, vậy tóm tắt ý tưởng thuật toán là thực hiện $N-1$ lượt việc đưa phần tử nhỏ nhất trong dãy hiện hành về vị trí đúng ở đầu dãy. Các bước tiến hành như sau:

Bước 1: $i = 1$;

Bước 2: Tìm phần tử $a[\min]$ nhỏ nhất trong dãy hiện hành từ $a[i]$ đến $a[N]$

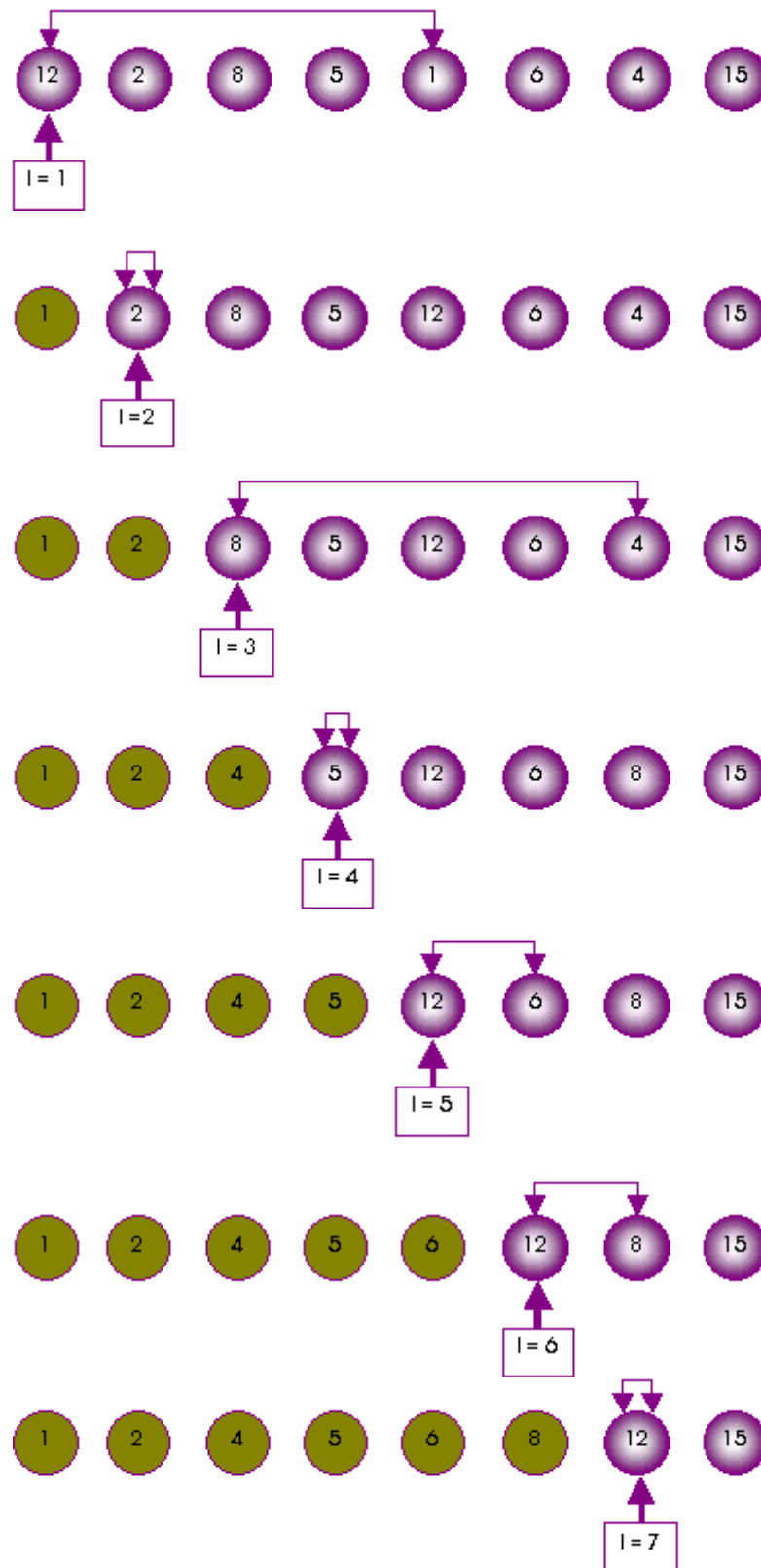
Bước 3: Hoán vị $a[\min]$ và $a[i]$

Bước 4: Nếu $i \leq N-1$ thì $i = i+1$; Lặp lại Bước 2

Ngừng lại: Dừng. // $N-1$ phần tử đã nằm đúng vị trí.

Ví dụ:

Cho dãy số a : 12 2 8 5 1 6 4 15



Cài đặt

Cài đặt thuật toán sắp xếp chọn trực tiếp thành hàm SelectionSort

```
void SelectionSort(int a[],int N )
{int min; // chỉ số phần tử nhỏ nhất trong dãy hiện hành
for (int i=0; i<N-1 ; i++)
{
    min = i;
```

```

        for(int j = i+1; j < N ; j++)
            if (a[j] < a[min])
                min = j; // ghi nhận vị trí phần tử hiện
nhỏ nhất
        Hoanvi(a[min], a[i]);
    }
}

```

Đánh giá giải thuật

Đối với giải thuật chọn trực tiếp, có thể thấy rằng ở lượt thứ i , bao giờ cũng cần $(n-i)$ lần so sánh để xác định phần tử nhỏ nhất hiện hành. Số lượng phép so sánh này không phụ thuộc vào tình trạng của dãy số ban đầu, do vậy trong mọi trường hợp có thể kết luận:

$$\text{Số lần so sánh} = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

Số lần hoán vị (một hoán vị bằng 3 phép gán) lại phụ thuộc vào tình trạng ban đầu của dãy số, ta chỉ có thể ước lượng trong từng trường hợp như sau:

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	$n(n-1)/2$	0
Xấu nhất	$n(n-1)/2$	$3n(n-1)/2$

2.2.3. Phương pháp chèn trực tiếp (Insertion Sort)

Giải thuật

Giả sử có một dãy a_1, a_2, \dots, a_n trong đó i phần tử đầu tiên a_1, a_2, \dots, a_{i-1} đã có thứ tự. Ý tưởng chính của giải thuật sắp xếp bằng phương pháp chèn trực tiếp là tìm cách chèn phần tử a_i vào vị trí thích hợp của đoạn đã được sắp để có dãy mới a_1, a_2, \dots, a_i trở nên có thứ tự. Vị trí này chính là vị trí giữa hai phần tử a_{k-1} và a_k thỏa $a_{k-1} < a_i < a_k$.

Cho dãy ban đầu a_1, a_2, \dots, a_n , ta có thể xem như đã có đoạn gồm một phần tử a_1 đã được sắp, sau đó thêm a_2 vào đoạn a_1 sẽ có đoạn a_1, a_2 được sắp; tiếp tục thêm a_3 vào đoạn a_1, a_2 để có đoạn a_1, a_2, a_3 được sắp; tiếp tục cho đến khi thêm xong a_n vào đoạn a_1, a_2, \dots, a_{n-1} sẽ có dãy a_1, a_2, \dots, a_n được sắp. Các bước tiến hành như sau:

Bước 1: $i = 2$; // giả sử có đoạn $a[1]$ đã được sắp

Bước 2: $x = a[i]$; Tìm vị trí pos thích hợp trong đoạn $a[1]$ đến $a[i-1]$ để chèn $a[i]$ vào

Bước 3: Dời chỗ các phần tử từ $a[pos]$ đến $a[i-1]$ sang phải 1 vị trí để dành chỗ cho $a[i]$

Bước 4: $a[pos] = x$; // có đoạn $a[1]..a[i]$ đã được sắp

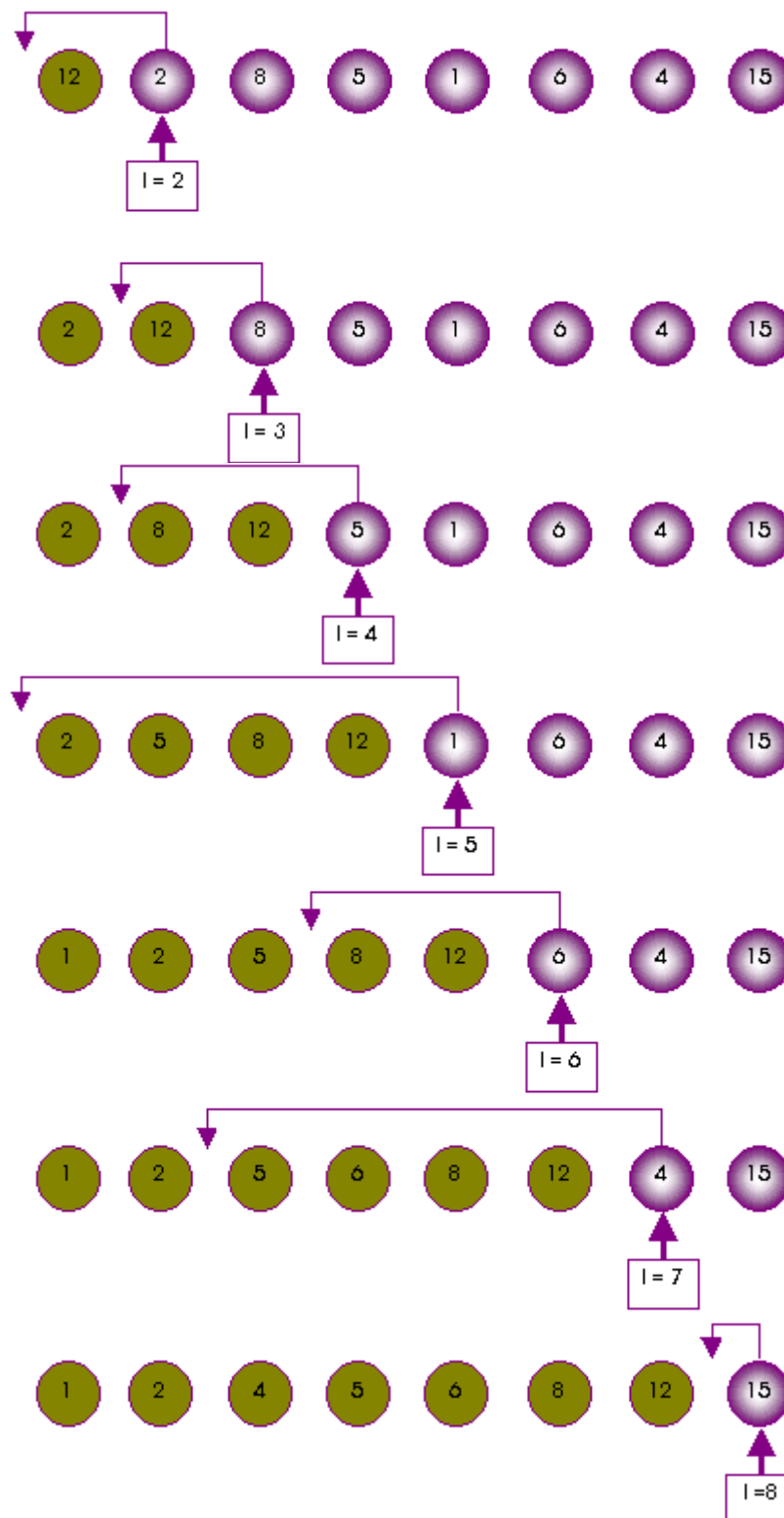
Bước 5: $i = i+1$;

Nếu $i < n$: Lặp lại Bước 2.

Ngược lại : Dừng.

Ví dụ

Cho dãy số a : 12 2 8 5 1 6 4 15



Cài đặt

Cài đặt thuật toán sắp xếp chèn trực tiếp thành hàm InsertionSort

```
void InsertionSort(int a[], int N )
```

```
{ int pos, i;
```

int x; //lưu giá trị a[i] tránh bị ghi đè khi dời chỗ các phần tử.

```
for(int i=1 ; i<N ; i++) //đoạn a[0] đã sắp
```





```

{
    x = a[i]; pos = i-1; // tìm vị trí chèn x
    while((pos >= 0) && (a[pos] > x))
        { // kết hợp dời chỗ các phần tử sẽ đứng sau x trong
        dãy mới
            a[pos+1] = a[pos];
            pos--;
        }
    a[pos+1] = x; // chèn x vào dãy
}
}

```

Đánh giá giải thuật

Đối với giải thuật chèn trực tiếp, các phép so sánh xảy ra trong mỗi vòng lặp while tìm vị trí thích hợp pos, và mỗi lần xác định vị trí đang xét không thích hợp, sẽ dời chỗ phần tử a[pos] tương ứng. Giải thuật thực hiện tất cả N-1 vòng lặp while, do số lượng phép so sánh và dời chỗ này phụ thuộc vào tình trạng của dãy số ban đầu, nên chỉ có thể ước lượng trong từng trường hợp như sau:

Trường hợp	Số phép so sánh	Số phép gán
Tốt nhất		
Xấu nhất		

2.2.4. Phương pháp đổi chỗ trực tiếp (Interchange Sort)

Giải thuật

Như đã đề cập ở đầu phần này, để sắp xếp một dãy số, ta có thể xét các nghịch thế có trong dãy và làm triệt tiêu dần chúng đi. Ý tưởng chính của giải thuật là xuất phát từ đầu dãy, tìm tất cả nghịch thế chứa phần tử này, triệt tiêu chúng bằng cách đổi chỗ phần tử này với phần tử tương ứng trong cặp nghịch thế. Lặp lại xử lý trên với các phần tử tiếp theo trong dãy. Các bước tiến hành như sau:

Bước 1 : $i = 1$; // bắt đầu từ đầu dãy

Bước 2 : $j = i+1$; // tìm các phần tử $a[j] < a[i]$, $j > i$

Bước 3 :

Trong khi $j \leq N$ thực hiện

Nếu $a[j] < a[i]$: $a[i] \leftrightarrow a[j]$; // xét cặp $a[i]$, $a[j]$

$j = j+1$;

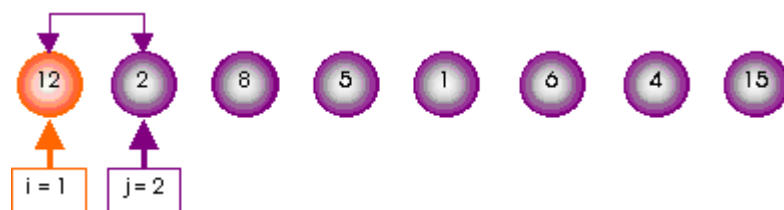
Bước 4 : $i = i+1$;

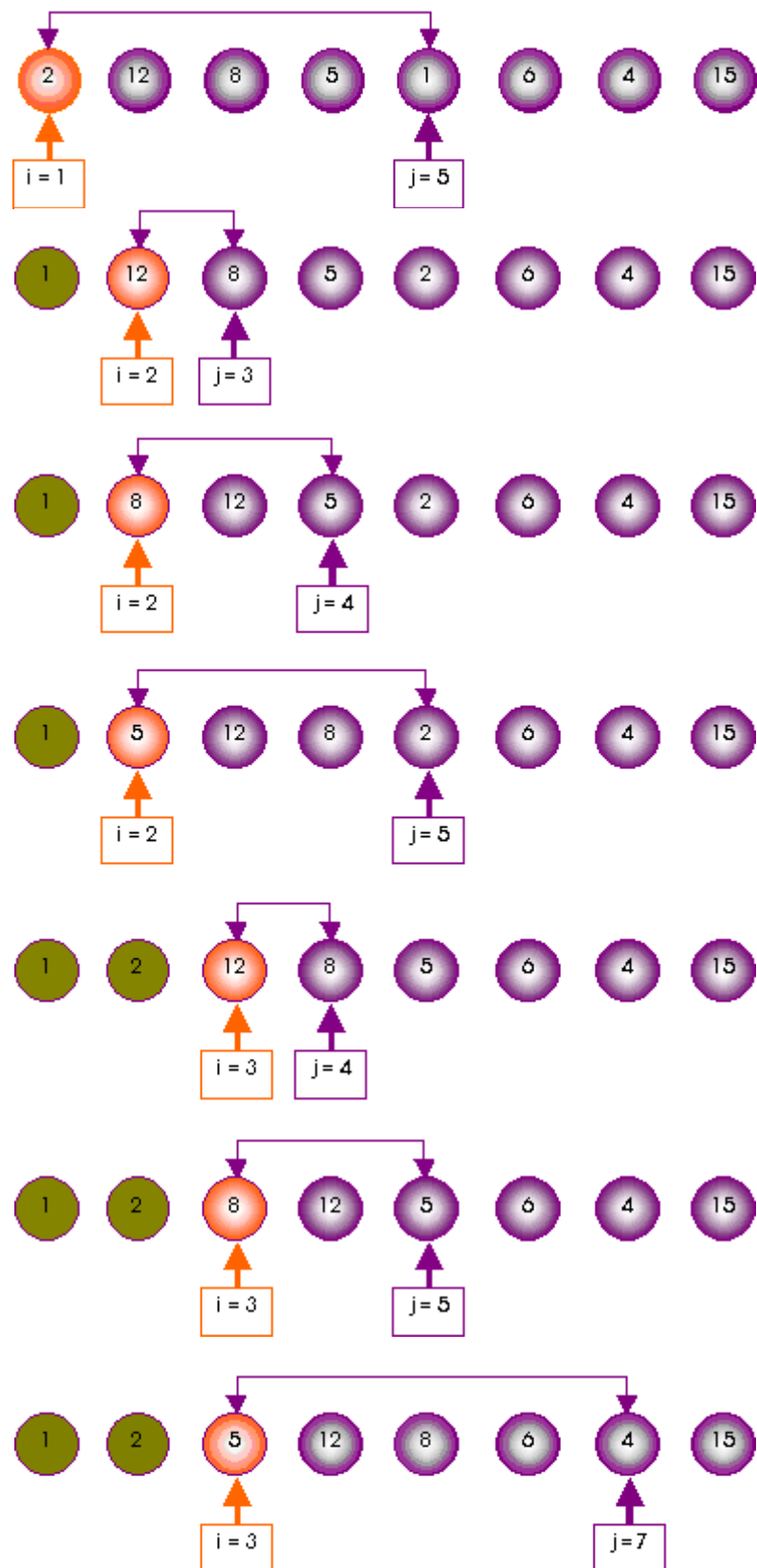
Nếu $i < n$: Lặp lại Bước 2.

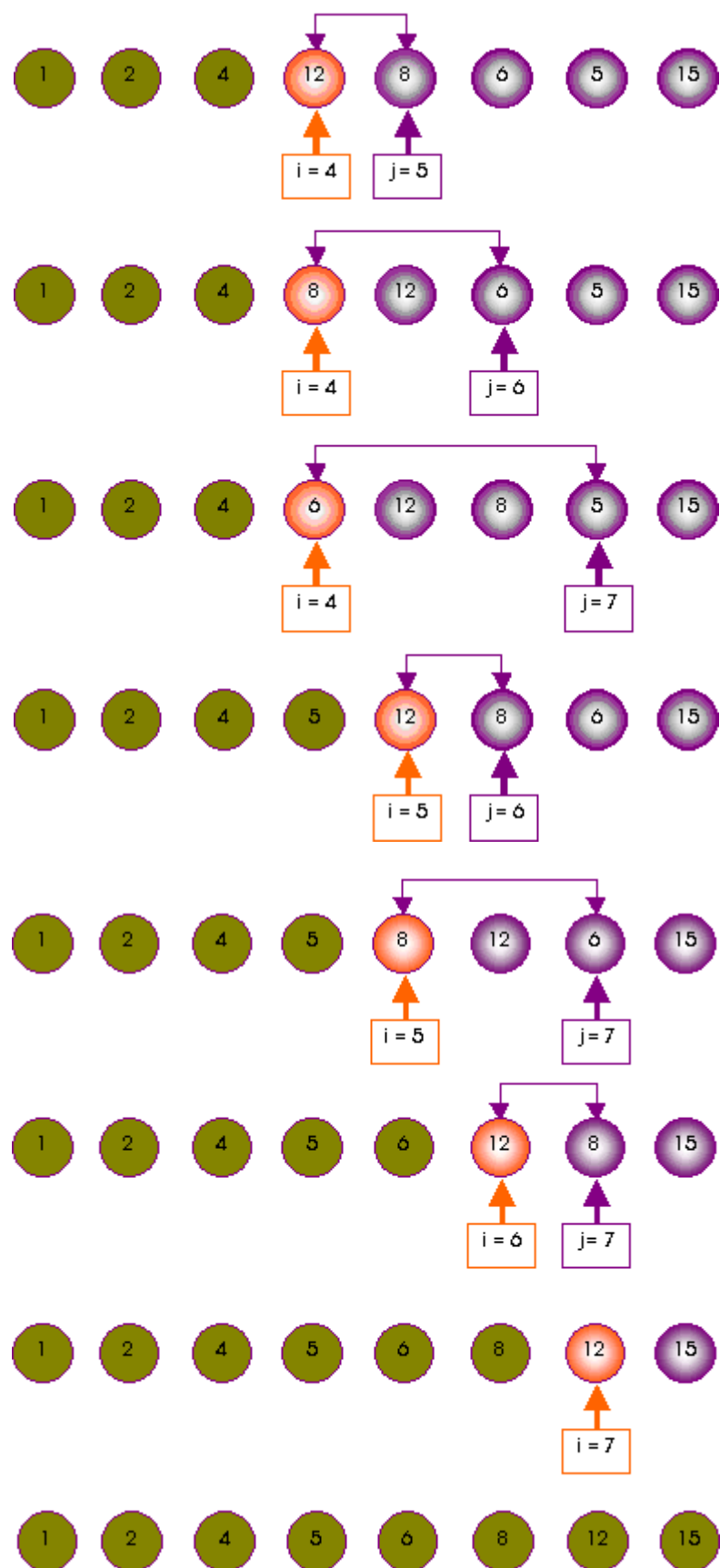
Ngược lại: Dừng.

Ví dụ

Cho dãy số a: 12 2 8 5 1 6 4 15










Cài đặt

Cài đặt thuật toán sắp xếp theo kiểu đổi chỗ trực tiếp thành hàm InterchangeSort:

```
void InterchangeSort(int a[], int N )
{
    int i, j;
    for (i = 1 ; i<N ; i++)
        for (j =i+1; j <= N ; j++)
            if(a[j ]< a[i])
                // nếu có sự sai vị trí thì đổi chỗ
                Hoanvi(a[i],a[j]);
}
```

Đánh giá giải thuật

Đối với giải thuật đổi chỗ trực tiếp, số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy số ban đầu, nhưng số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh, có thể ước lượng trong từng trường hợp như sau:

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất		0
Xấu nhất		

2.2.5. Phương pháp nổi bọt (Bubble Sort)

Giải thuật

Ý tưởng chính của giải thuật là xuất phát từ cuối (đầu) dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ (lớn) hơn trong cặp phần tử đó về vị trí đúng đầu (cuối) dãy hiện hành, sau đó sẽ không xét đến nó ở bước tiếp theo, do vậy ở lần xử lý thứ i sẽ có vị trí đầu dãy là i . Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào để xét. Các bước tiến hành như sau:

Bước 1 : $i = 1$; // lần xử lý đầu tiên

Bước 2 : $j = N$; //Duyệt từ cuối dãy ngược về vị trí i

Trong khi ($j < i$) thực hiện:

Nếu $a[j]<a[j-1]$: $a[j] \leftrightarrow a[j-1]$; //xét cặp phần tử kế cận

$j = j-1$;

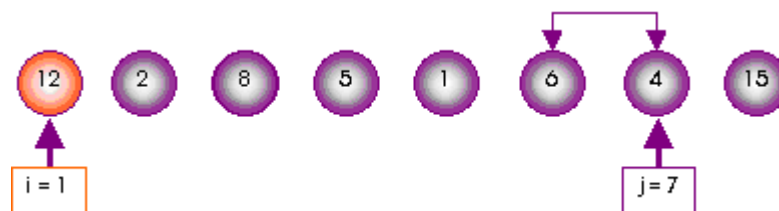
Bước 3 : $i = i+1$; // lần xử lý kế tiếp

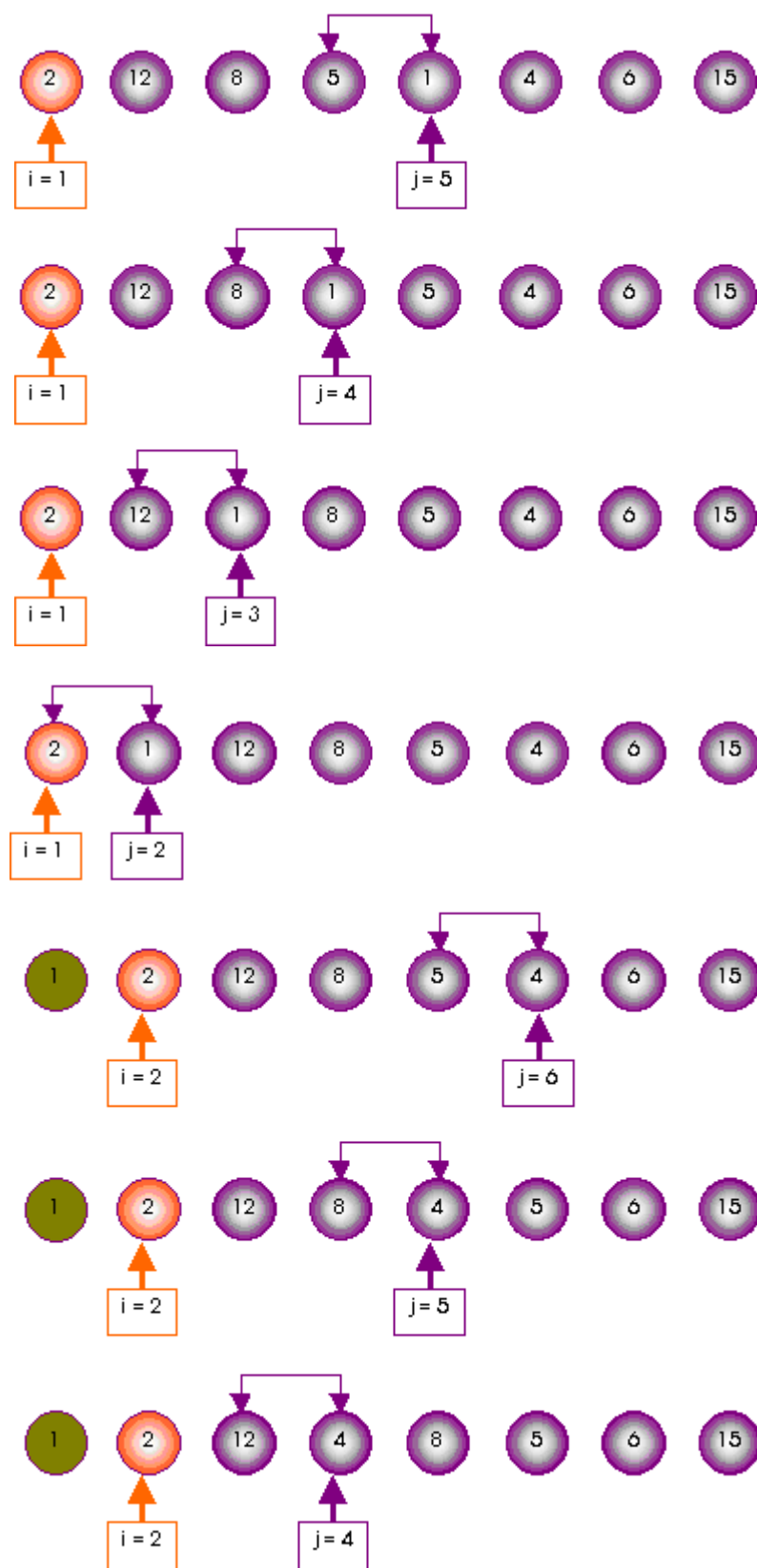
Nếu $i > N-1$: Hết dãy. Dừng

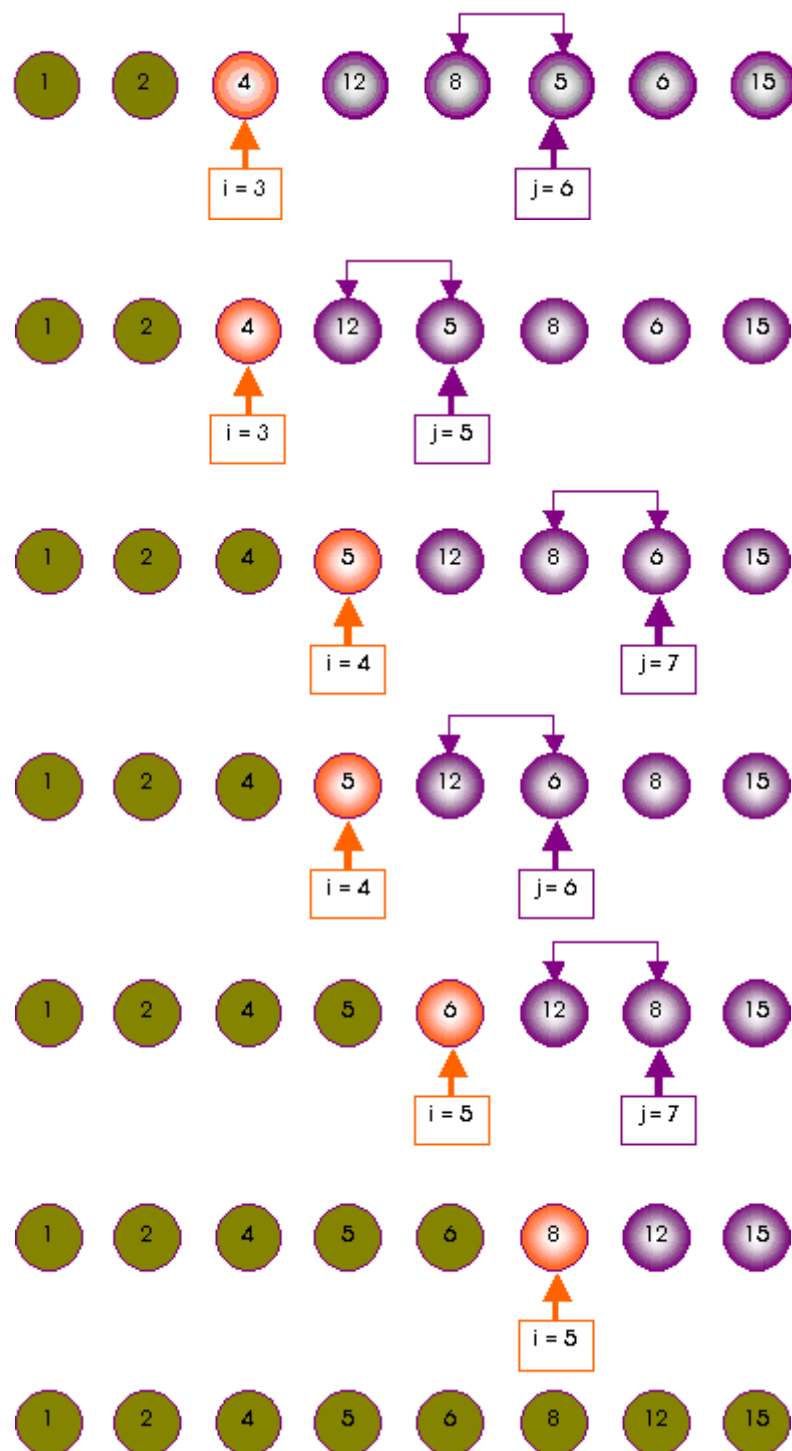
Ngược lại : Lặp lại Bước 2.

Ví dụ

Cho dãy số a: 12 2 8 5 1 6 4 15










Cài đặt

Cài đặt thuật toán sắp xếp theo kiểu nổi bọt thành hàm BubbleSort:

```
void BubleSort(int a[], int N )
{
    int i, j;
    for (i = 1 ; i<N-1 ; i++)
        for (j =N-1; j >i ; j --)
            if(a[j]< a[j-1])
                // nếu sai vị trí thì đổi chỗ
                Hoanvi(a[j],a[j-1]);
}
```

Đánh giá giải thuật

Đối với giải thuật nổi bọt, số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy số ban đầu, nhưng số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh, có thể ước lượng trong từng trường hợp như sau:

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất		0
Xấu nhất		

Nhận xét

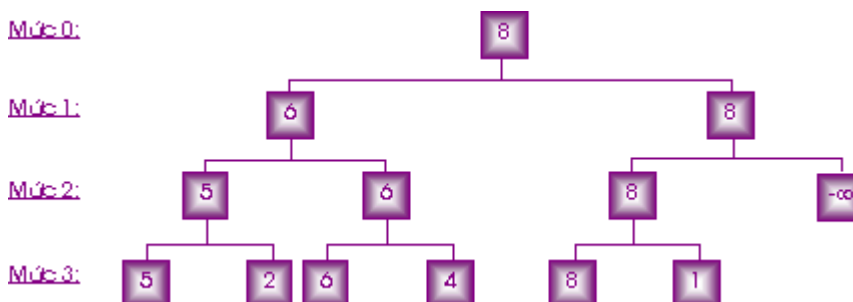
BubbleSort có các khuyết điểm sau: không nhận diện được tình trạng dãy đã có thứ tự hay có thứ tự từng phần. Các phần tử nhỏ được đưa về vị trí đúng rất nhanh, trong khi các phần tử lớn lại được đưa về vị trí đúng rất chậm.

2.2.6. Phương pháp sắp xếp cây (Heap Sort)

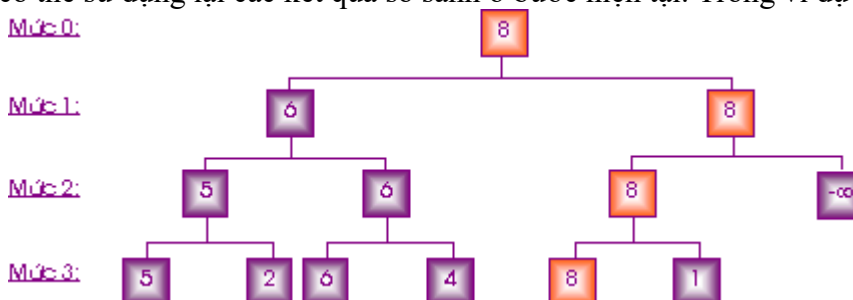
2.2.6.1. Giải thuật Sắp xếp cây

Khi tìm phần tử nhỏ nhất ở bước i , phương pháp sắp xếp chọn trực tiếp không tận dụng được các thông tin đã có được do các phép so sánh ở bước $i-1$. Vì lý do trên người ta tìm cách xây dựng một thuật toán sắp xếp có thể khắc phục nhược điểm này.

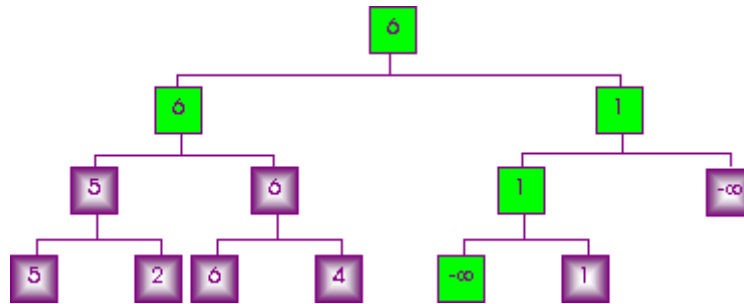
Mấu chốt để giải quyết vấn đề vừa nêu là phải tìm ra được một cấu trúc dữ liệu cho phép tích lũy các thông tin về sự so sánh giá trị các phần tử trong quá trình sắp xếp. Giả sử dữ liệu cần sắp xếp là dãy số : 5 2 6 4 8 1 được bố trí theo quan hệ so sánh và tạo thành sơ đồ dạng cây như sau:



Trong đó một phần tử ở mức i chính là phần tử lớn trong cặp phần tử ở mức $i+1$, do đó phần tử ở mức 0 (nút gốc của cây) luôn là phần tử lớn nhất của dãy. Nếu loại bỏ phần tử gốc ra khỏi cây (nghĩa là đưa phần tử lớn nhất về đúng vị trí), thì việc cập nhật cây chỉ xảy ra trên những nhánh liên quan đến phần tử mới loại bỏ, còn các nhánh khác được bảo toàn, nghĩa là bước kế tiếp có thể sử dụng lại các kết quả so sánh ở bước hiện tại. Trong ví dụ trên ta có:



Loại bỏ 8 ra khỏi cây và thế vào các chỗ trống giá trị $-\infty$ để tiện việc cập nhật lại cây:



Có thể nhận thấy toàn bộ nhánh trái của gốc 8 cũ được bảo toàn, do vậy bước kế tiếp để chọn được phần tử lớn nhất hiện hành là 6, chỉ cần làm thêm một phép so sánh 1 với 6.

Tiến hành nhiều lần việc loại bỏ phần tử gốc của cây cho đến khi tất cả các phần tử của cây đều là $-\infty$, khi đó xếp các phần tử theo thứ tự loại bỏ trên cây sẽ có dãy đã sắp xếp. Trên đây là ý tưởng của giải thuật sắp xếp cây.

2.2.6.2. Cấu trúc dữ liệu Heap

Tuy nhiên, để cài đặt thuật toán này một cách hiệu quả, cần phải tổ chức một cấu trúc lưu trữ dữ liệu có khả năng thể hiện được quan hệ của các phần tử trong cây với n ô nhớ thay vì $2n-1$ như trong ví dụ. Khái niệm heap và phương pháp sắp xếp Heapsort do J.Williams đề xuất đã giải quyết được các khó khăn trên.

Định nghĩa Heap:

Giả sử xét trường hợp sắp xếp tăng dần, khi đó Heap được định nghĩa là một dãy các phần tử a_1, a_2, \dots, a_r thỏa các quan hệ sau với mọi $i \in [1, r]$:

$$1 \quad a_i \geq a_{2i}$$

$$2 \quad a_i \geq a_{2i+1} \quad \{(a_i, a_{2i}), (a_i, a_{2i+1}) \text{ là các cặp phần tử liên đới} \}$$

Heap có các tính chất sau:

Tính chất 1: Nếu a_1, a_2, \dots, a_r là một heap thì khi cắt bỏ một số phần tử ở hai đầu của heap, dãy còn lại vẫn là một heap.

Tính chất 2: Nếu a_1, a_2, \dots, a_n là một heap thì phần tử a_1 (đầu heap) luôn là phần tử lớn nhất trong heap.

Tính chất 3: Mọi dãy a_1, a_2, \dots, a_r với $2l > r$ là một heap.

Giải thuật Heapsort:

Giải thuật Heapsort trải qua 2 giai đoạn:

Giai đoạn 1: Hiệu chỉnh dãy số ban đầu thành heap;

Giai đoạn 2: Sắp xếp dãy số dựa trên heap:

Bước 1: Đưa phần tử nhỏ nhất về vị trí đứng ở cuối dãy: $r = n$; Hoán vị (a_1, a_r) ;

Bước 2: Loại bỏ phần tử nhỏ nhất ra khỏi heap: $r = r-1$; Hiệu chỉnh phần còn lại của dãy từ a_1, a_2, \dots, a_r thành một heap.

Bước 3: Nếu $r > 1$ (heap còn phần tử): Lặp lại Bước 2

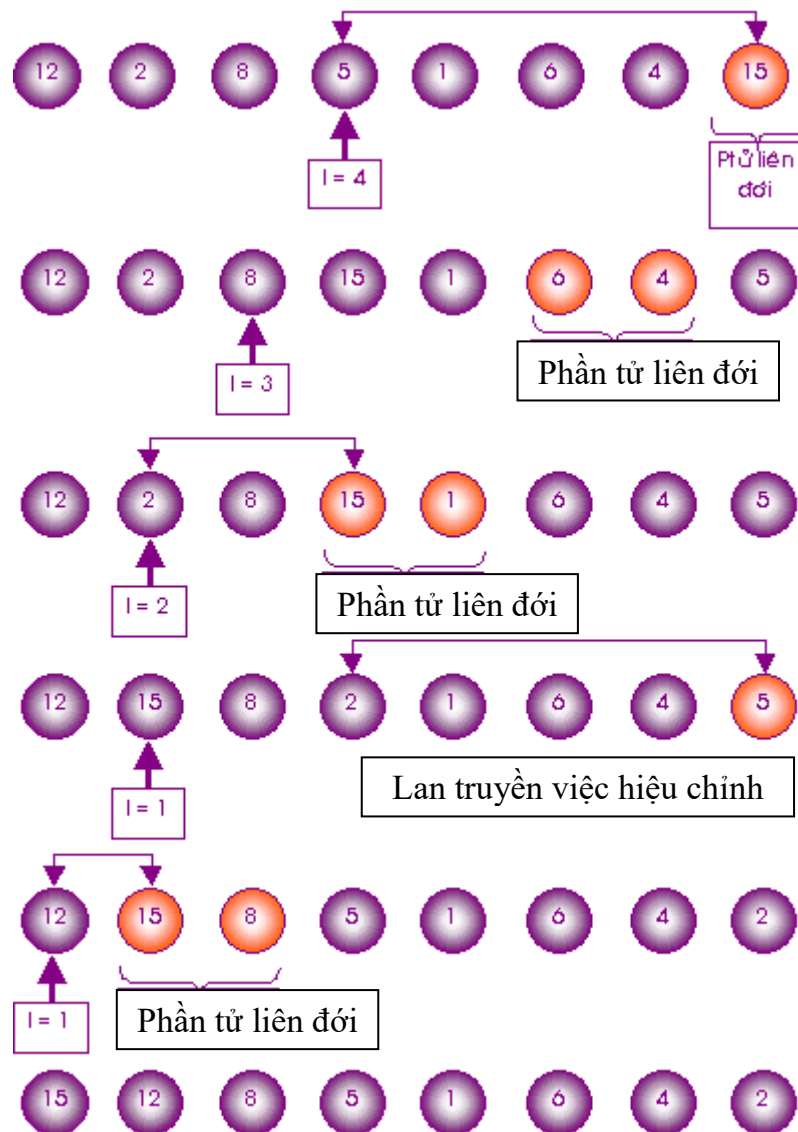
Ngược lại: Dừng

Dựa trên tính chất 3, ta có thể thực hiện giai đoạn 1 bằng cách bắt đầu từ heap mặc nhiên $a_{n/2+1}, a_{n/2+2}, \dots, a_n$, lần lượt thêm vào các phần tử $a_{n/2}, a_{n/2-1}, \dots, a_1$ ta sẽ nhận được heap theo mong muốn. Như vậy, giai đoạn 1 tương đương với $n/2$ lần thực hiện bước 2 của giai đoạn 2.

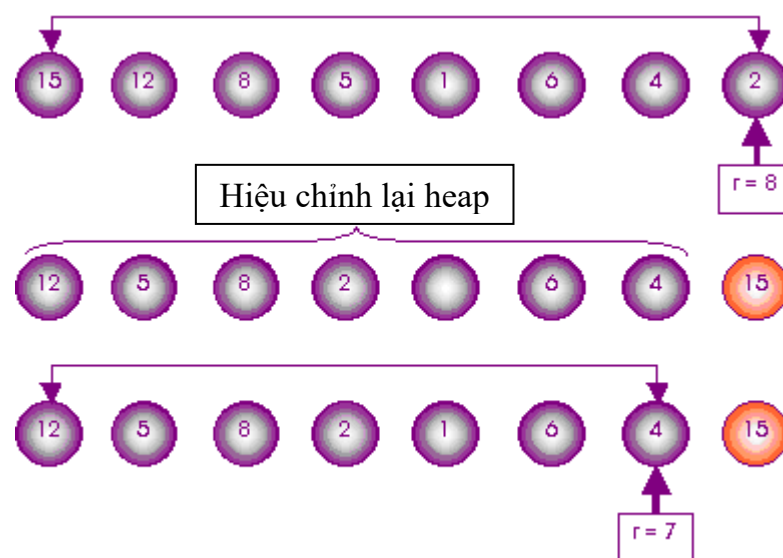
Ví dụ

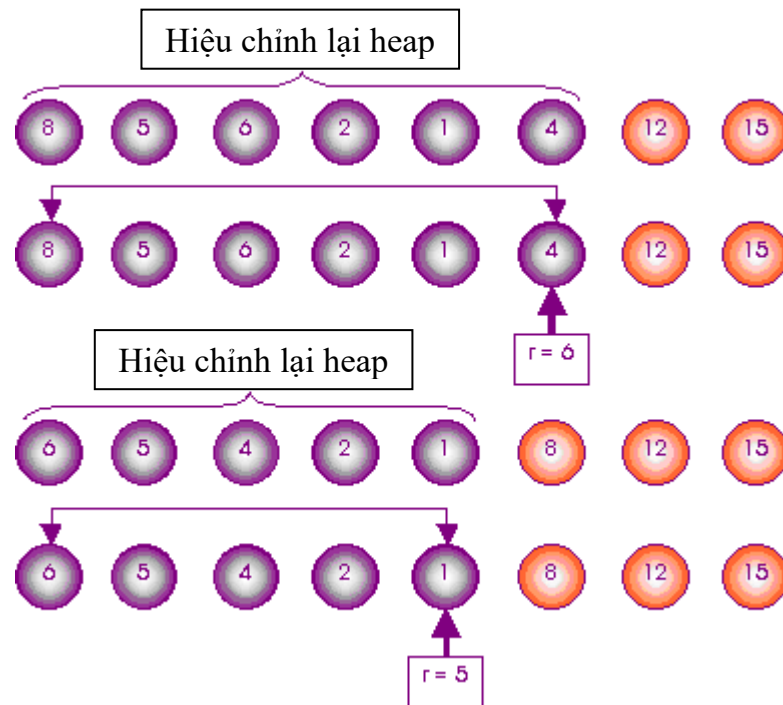
Cho dãy số a : 12 2 8 5 1 6 4 15

Giai đoạn 1: hiệu chỉnh dãy ban đầu thành heap



Giai đoạn 2: Sắp xếp dãy số dựa trên heap





thực hiện tương tự cho $r=5, 4, 3, 2$ ta được:



Cài đặt

Để cài đặt giải thuật Heapsort cần xây dựng các thủ tục phụ trợ:

*** Thủ tục hiệu chỉnh dãy $a_1, a_{l+1} \dots a_r$ thành heap:**

Giả sử có dãy $a_1, a_{l+1} \dots a_r$, trong đó đoạn $a_{l+1} \dots a_r$ đã là một heap. Ta cần xây dựng hàm hiệu chỉnh $a_1, a_{l+1} \dots a_r$ thành heap. Để làm điều này, ta lần lượt xét quan hệ của một phần tử a_i nào đó với các phần tử liên đới của nó trong dãy là a_{2i} và a_{2i+1} , nếu vi phạm điều kiện quan hệ của heap, thì đổi chỗ a_i với phần tử liên đới thích hợp của nó. Lưu ý việc đổi chỗ này có thể gây phản ứng dây chuyền:

```
void Shift (int a[ ], int l, int r )
{
    int x, i, j;
    i = l;
    j = 2*i; // (ai , aj), (ai , aj+1) là các phần tử liên đới
    x = a[i];
    while (j<=r)
    {
        if (j<r) // nếu có đủ 2 phần tử liên đới
            if (a[j]<a[j+1])
                // xác định phần tử liên đới lớn nhất
                j = j+1;
        if (a[j]<x)
            break; //thoả quan hệ liên đới, dùng
        else
        {
            a[i] = a[j];
            i = j;
        }
        // xét tiếp khả năng hiệu chỉnh lan truyền
        j = 2*i;
    }
}
```

```

        a[i] = x;
    }
}

```

*** Hiệu chỉnh dãy $a_1, a_2 \dots a_N$ thành heap :**

Cho một dãy bất kỳ a_1, a_2, \dots, a_r , theo tính chất 3, ta có dãy $a_{n/2+1}, a_{n/2+2} \dots a_n$ đã là một heap. Ghép thêm phần tử $a_{n/2}$ vào bên trái heap hiện hành và hiệu chỉnh lại dãy $a_{n/2}, a_{n/2+1}, \dots, a_r$ thành heap,

```

void CreateHeap(int a[], int N )
{
    int l;
    l = N/2; // a[l] là phần tử ghép thêm
    while (l > 0)
    {
        Shift(a, l, N);
        l = l - 1;
    }
}

```

Khi đó hàm Heapsort có dạng sau:

```

void HeapSort (int a[], int N)
{
    int r;
    CreateHeap(a, N)
    r = N; // r là vị trí đúng cho phần tử nhỏ nhất
    while (r > 0)
    {
        Hoanvi(a[l], a[r]);
        r = r - 1;
        Shift(a, l, r);
    }
}

```

Đánh giá giải thuật

Việc đánh giá giải thuật Heapsort rất phức tạp, nhưng đã chứng minh được trong trường hợp xấu nhất độ phức tạp $\sim O(n \log_2 n)$

2.2.7. Phương pháp sắp xếp phân hoạch (Quick Sort)

Để sắp xếp dãy a_1, a_2, \dots, a_n giải thuật QuickSort dựa trên việc phân hoạch dãy ban đầu thành hai phần:

Dãy con 1: Gồm các phần tử $a_1 \dots a_i$ có giá trị không lớn hơn x

Dãy con 2: Gồm các phần tử $a_i \dots a_n$ có giá trị không nhỏ hơn x với x là giá trị của một phần tử tùy ý trong dãy ban đầu. Sau khi thực hiện phân hoạch, dãy ban đầu được phân thành 3 phần:

1. $a_k < x$, với $k = 1..i$
2. $a_k = x$, với $k = i..j$
3. $a_k > x$, với $k = j..N$

$a_k < x$	$a_k = x$	$a_k > x$
-----------	-----------	-----------

Trong đó dãy con thứ 2 đã có thứ tự, nếu các dãy con 1 và 3 chỉ có 1 phần tử thì chúng cũng đã có thứ tự, khi đó dãy ban đầu đã được sắp. Ngược lại, nếu các dãy con 1 và 3 có nhiều hơn 1 phần tử thì dãy ban đầu chỉ có thứ tự khi các dãy con 1, 3 được sắp. Để sắp xếp dãy con 1 và 3, ta lần lượt tiến hành việc phân hoạch từng dãy con theo cùng phương pháp phân hoạch dãy ban đầu vừa trình bày.

Giải thuật phân hoạch dãy a_1, a_{l+1}, \dots, a_r thành 2 dãy con:

Bước 1 :

Chọn tùy ý một phần tử $a[k]$ trong dãy là giá trị mốc, $1 \leq k \leq r$:
 $x = a[k]; i = l; j = r$;

Bước 2:

Phát hiện và hiệu chỉnh cặp phần tử $a[i], a[j]$ nằm sai chỗ :

Bước 2a:

Trong khi $(a[i] < x) i++$;

Bước 2b:

Trong khi $(a[j] > x) j--$;

Bước 2c:

Nếu $i < j$ // $a[i] > a[j]$ mà $a[j]$ đứng sau $a[i]$
Hoán vị $(a[i], a[j])$;

Bước 3 :

Nếu $i < j$: Lặp lại Bước 2.//chưa xét hết mảng

Nếu $i > j$: Dừng

Nhận xét:

Về nguyên tắc, có thể chọn giá trị mốc x là một phần tử tùy ý trong dãy, nhưng để đơn giản, dễ diễn đạt giải thuật, phần tử có vị trí giữa thường được chọn, khi đó $k = (l + r) / 2$.

Giá trị mốc x được chọn sẽ có tác động đến hiệu quả thực hiện thuật toán vì nó quyết định số lần phân hoạch. Số lần phân hoạch sẽ ít nhất nếu ta chọn được x là phần tử median của dãy. Tuy nhiên do chi phí xác định phần tử median quá cao nên trong thực tế người ta không chọn phần tử này mà chọn phần tử nằm chính giữa dãy làm mốc với hy vọng nó có thể gần với giá trị median

Giải thuật phân hoạch dãy sắp xếp dãy a_l, a_{l+1}, \dots, a_r :

Có thể phát biểu giải thuật sắp xếp QuickSort một cách đệ quy như sau:

Bước 1 : Phân hoạch dãy $a_l \dots a_r$ thành các dãy con:

- Dãy con 1 : $a_l \dots a_j < x$
- Dãy con 2 : $a_{j+1} \dots a_{i-1} = x$
- Dãy con 3 : $a_i \dots a_r > x$

Bước 2 :

Nếu $(1 < j)$ // dãy con 1 có nhiều hơn 1 phần tử

Phân hoạch dãy $a_l \dots a_j$

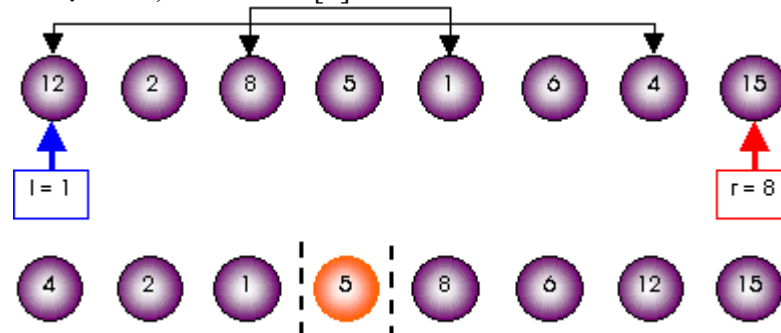
Nếu $(i < r)$ // dãy con 3 có nhiều hơn 1 phần tử

Phân hoạch dãy $a_i \dots a_r$

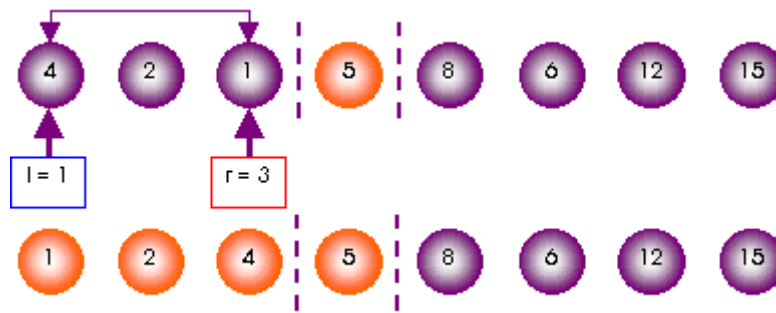
Ví dụ

Cho dãy số a : 12 2 8 5 1 6 4 15

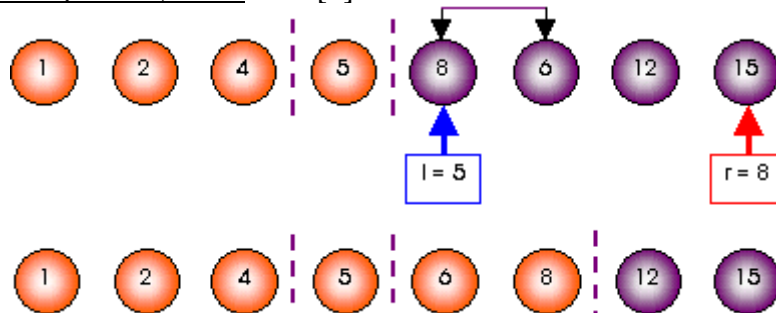
Phân hoạch đoạn $l = 1, r = 8$: $x = a[4] = 5$



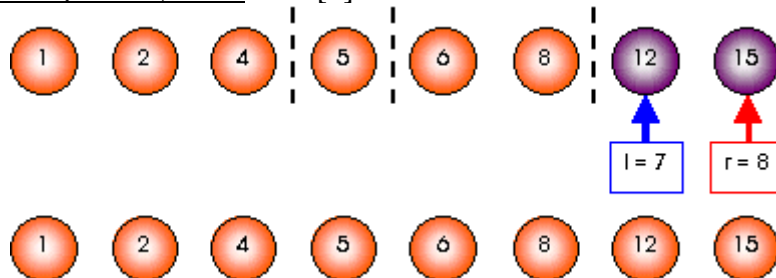
Phân hoạch đoạn $l = 1, r = 3$: $x = a[2] = 2$



Phân hoạch đoạn $l = 5, r = 8$: $x = a[6] = 6$



Phân hoạch đoạn $l = 7, r = 8$: $x = a[7] = 6$



Dừng.

Cài đặt

Thuật toán QuickSort có thể được cài đặt đệ qui như sau :

```
void QuickSort(int a[], int l, int r)
{
    int i, j;
    int x;
    x = a[(l+r)/2];          // chọn phần tử giữa làm giá trị mốc
    i = l; j = r;
    do {
        while(a[i] < x) i++;
        while(a[j] > x) j--;
        if(i <= j)
        {
            Hoanvi(a[i], a[j]);
            i++; j--;
        }
    } while(i < j);
    if(l < j)
        QuickSort(a, l, j);
    if(i < r)
        QuickSort(a, i, r);
}
```

Đánh giá giải thuật

Hiệu quả thực hiện của giải thuật QuickSort phụ thuộc vào việc chọn giá trị mốc. Trường hợp tốt nhất xảy ra nếu mỗi lần phân hoạch đều chọn được phần tử median (phần tử lớn hơn (hay bằng) nửa số phần tử, và nhỏ hơn (hay bằng) nửa số phần tử còn lại) làm mốc, khi đó dãy được phân chia thành 2 phần bằng nhau và cần $\log_2(n)$ lần phân hoạch thì sắp xếp xong. Nhưng nếu mỗi lần phân hoạch lại chọn nhầm phần tử có giá trị cực đại (hay cực tiểu) là mốc, dãy sẽ bị phân chia thành 2 phần không đều: một phần chỉ có 1 phần tử, phần còn lại gồm $(n-1)$ phần tử, do vậy cần phân hoạch n lần mới sắp xếp xong. Ta có bảng tổng kết.

Trường hợp	Độ phức tạp
Tốt nhất	$n \cdot \log(n)$
Trung bình	$n \cdot \log(n)$
Xấu nhất	n^2

❖ Bài tập củng cố:

1. Xét mảng các số nguyên có nội dung như sau:

-9 -9 -5 -2 0 3 7 7 10 15

- Tính số lần so sánh để tìm ra phần tử $X = -9$ bằng phương pháp: tìm tuyến tính và tìm nhị phân. Nhận xét và so sánh 2 phương pháp tìm nêu trên trong trường hợp này và trong trường hợp tổng quát.
- Trong trường hợp tìm nhị phân, phần tử nào sẽ được tìm thấy (thứ 1 hay 2). Xây dựng thuật toán tìm phần tử nhỏ nhất (lớn nhất) trong một mảng các số nguyên.

2. Cài đặt các thuật toán tìm kiếm đã trình bày. Thể hiện trực quan các thao tác của thuật toán.

- Hãy viết hàm tìm tất cả các số nguyên tố nằm trong mảng một chiều a có n phần tử.
- Hãy viết hàm tìm dãy con tăng dài nhất của mảng một chiều a có n phần tử (dãy con là một dãy liên tiếp các phần của a).
- Cài đặt thuật toán tìm phần tử trung vị (median) của một dãy số.

3. Trong 3 phương pháp sắp xếp cơ bản (chọn trực tiếp, chèn trực tiếp, nổi bọt) phương pháp nào thực hiện sắp xếp nhanh nhất với một dãy đã có thứ tự? Giải thích.

4. Cho dãy số 5 1 2 8 4 7 0 12 4 3 24 1 4, hãy minh họa kết quả sắp xếp dãy số này từng bước với 6 giải thuật đã học.

5. Cài đặt các thuật toán sắp xếp đã trình bày. Thể hiện trực quan các thao tác của thuật toán. Cài đặt thêm chức năng xuất bảng lương nhân viên theo thứ tự tiền lương tăng dần cho bài tập 5 chương 1.

CHƯƠNG 3

DANH SÁCH LIÊN KẾT

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể:

- Hiểu được định nghĩa và cách tổ chức danh sách liên kết.
- Vận dụng và cài đặt được các phép toán trên danh sách liên kết đơn.
- Vận dụng và cài đặt được các phép toán trên danh sách liên kết kép.

3.1. DANH SÁCH LIÊN KẾT (LINK LIST)

3.1.1. Định nghĩa:

Cho T là một kiểu được định nghĩa trước, kiểu danh sách T_x gồm các phần tử thuộc kiểu T được định nghĩa là:

$$T_x = \langle V_x, O_x \rangle$$

Trong đó: $V_x = \{\text{tập hợp có thứ tự các phần tử kiểu } T \text{ được móc nối với nhau theo trình tự tuyến tính}\}$; $O_x = \{\text{Tạo danh sách; Tìm 1 phần tử trong danh sách; Chèn một phần tử vào danh sách; Huỷ một phần tử khỏi danh sách; Liệt kê danh sách, Sắp xếp danh sách ...}\}$

Ví dụ: Hồ sơ các học sinh của một trường được tổ chức thành danh sách gồm nhiều hồ sơ của từng học sinh; số lượng học sinh trong trường có thể thay đổi do vậy cần có các thao tác thêm, hủy một hồ sơ; để phục vụ công tác giáo vụ cần thực hiện các thao tác tìm hồ sơ của một học sinh, in danh sách hồ sơ ...

3.1.2. Các hình thức tổ chức danh sách

Có nhiều hình thức tổ chức mỗi liên hệ tuần tự giữa các phần tử trong cùng một danh sách:

Mỗi liên hệ giữa các phần tử được thể hiện ngầm: mỗi phần tử trong danh sách được đặc trưng bằng chỉ số. Cặp phần tử x_i, x_{i+1} được xác định là kế cận trong danh sách nhờ vào quan hệ giữa cặp chỉ số i và $(i+1)$. Với hình thức tổ chức này, các phần tử của danh sách thường bắt buộc phải lưu trữ liên tiếp trong bộ nhớ để có thể xây dựng công thức xác định địa chỉ phần tử thứ i :

1	2	3	4	5
9	4	5	3	8

$$\text{address}(i) = \text{address}(1) + (i-1) * \text{sizeof}(T)$$

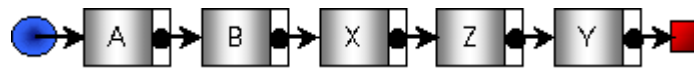
Có thể xem mảng và tập tin là những danh sách đặc biệt được tổ chức theo hình thức liên kết "ngầm" giữa các phần tử. Tuy nhiên mảng có một đặc trưng giới hạn là số phần tử mảng cố định, do vậy không có thao tác thêm, hủy trên mảng; trường hợp tập tin thì các phần tử được lưu trữ trên bộ nhớ phụ có những đặc tính lưu trữ riêng sẽ được trình bày chi tiết ở giáo trình Cấu trúc dữ liệu 2.

Cách biểu diễn này cho phép truy xuất ngẫu nhiên, đơn giản và nhanh chóng đến một phần tử bất kỳ trong danh sách, nhưng lại hạn chế về mặt sử dụng bộ nhớ. Đối với mảng, số phần tử được xác định trong thời gian biên dịch và cần cấp phát vùng nhớ liên tục. Trong trường hợp tổng kích thước bộ nhớ trống còn đủ để chứa toàn bộ mảng nhưng các ô nhớ trống lại không nằm kế cận nhau thì cũng không cấp phát vùng nhớ cho mảng được. Ngoài ra do kích thước mảng cố định mà số phần tử của danh sách lại khó dự trù chính xác nên có thể gây ra tình trạng thiếu hụt hay lãng phí bộ nhớ. Hơn nữa các thao tác thêm, hủy một phần tử vào danh sách được thực hiện không tự nhiên trong hình thức tổ chức này.

Mỗi liên hệ giữa các phần tử được thể hiện tường minh: mỗi phần tử ngoài các thông tin về bản thân còn chứa một liên kết (địa chỉ) đến phần tử kế trong danh sách nên còn được gọi là danh sách móc nối. Do liên kết tường minh, với hình thức này các phần tử trong danh sách không cần phải lưu trữ kế cận trong bộ nhớ nên khắc phục được các khuyết điểm của

hình thức tổ chức mảng, nhưng việc truy xuất đến một phần tử đòi hỏi phải thực hiện truy xuất qua một số phần tử khác. Có nhiều kiểu tổ chức liên kết giữa các phần tử trong danh sách như :

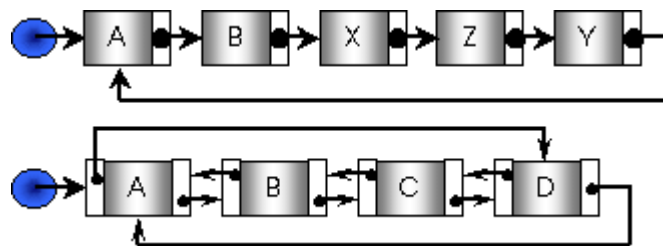
Danh sách liên kết đơn: mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách:



Danh sách liên kết kép: mỗi phần tử liên kết với các phần tử đứng trước và sau nó trong danh sách:



Danh sách liên kết vòng: phần tử cuối danh sách liên kết với phần tử đầu danh sách:



Hình thức liên kết này cho phép các thao tác thêm, hủy trên danh sách được thực hiện dễ dàng, phản ánh được bản chất linh động của danh sách.

3.2. DANH SÁCH LIÊN KẾT ĐƠN

3.2.1. Tổ chức danh sách liên kết đơn

Cấu trúc dữ liệu của một phần tử trong danh sách đơn:

Mỗi phần tử của danh sách đơn là một cấu trúc chứa 2 thông tin:

Thành phần dữ liệu: lưu trữ các thông tin về bản thân phần tử.

Thành phần mỗi liên kết: lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh sách.

Ta có định nghĩa tổng quát

```

typedef struct NODE
{
    Data Info; // Data là kiểu đã định nghĩa trước
    NODE* pNext; // con trỏ chỉ đến cấu trúc node
}NODE;
  
```

Ví dụ : Định nghĩa danh sách đơn lưu trữ hồ sơ sinh viên:

```

typedef struct SinhVien
{
    char Ten[30];
    int MaSV;
}SinhVien;
typedef struct SVNode
{
    SV Info;
    SVNode* pNext;
}SVNode;
  
```

Một phần tử trong danh sách đơn là một biến động sẽ được yêu cầu cấp phát khi cần. Và danh sách đơn chính là sự liên kết các biến động này với nhau do vậy đạt được sự linh

động khi thay đổi số lượng các phần tử.

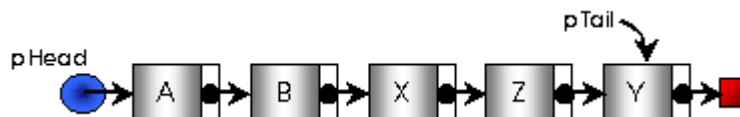
Nếu biết được địa chỉ của phần tử đầu tiên trong danh sách đơn thì có thể dựa vào thông tin pNext của nó để truy xuất đến phần tử thứ 2 trong danh sách, và lại dựa vào thông tin Next của phần tử thứ 2 để truy xuất đến phần tử thứ 3...nghĩa là để quản lý một danh sách đơn chỉ cần biết địa chỉ phần tử đầu danh sách. Thường một con trỏ Head sẽ được dùng để lưu trữ địa chỉ phần tử đầu danh sách, ta gọi Head là đầu danh sách. Ta có khai báo:

```
NODE *pHead;
```

Tuy về nguyên tắc chỉ cần quản lý danh sách thông qua đầu danh sách pHead, nhưng thực tế có nhiều trường hợp cần làm việc với phần tử cuối danh sách, khi đó mỗi lần muốn xác định phần tử cuối danh sách lại phải duyệt từ đầu danh sách. Để tiện lợi, có thể sử dụng thêm một con trỏ pTail giữ địa chỉ phần tử cuối danh sách. Khai báo pTail như sau:

```
NODE *pTail;
```

Lúc này có danh sách đơn:



3.2.2. Các thao tác cơ bản trên danh sách liên kết đơn

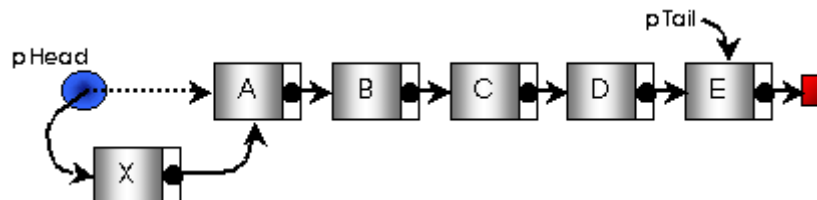
Giả sử có các định nghĩa:

```
typedef struct Node
{
    Data Info;
    Node* pNext;
}Node;
typedef struct List
{
    Node* pHead;
    Node* pTail;
}List;
```

Node *new_ele // giữ địa chỉ của một phần tử mới được tạo

Data x; // lưu thông tin về một phần tử sẽ được tạo

3.2.2.1. Chèn một phần tử vào danh sách:



Có 3 loại thao tác chèn new_ele vào danh sách:

Chèn vào đầu danh sách

Thuật toán:

Bắt đầu:

Nếu Danh sách rỗng Thì

Bước 1.1 : Head = new_elelment;

Bước 1.2 : Tail = Head;

Ngược lại

Bước 2.1 : new_ele ->pNext = Head;

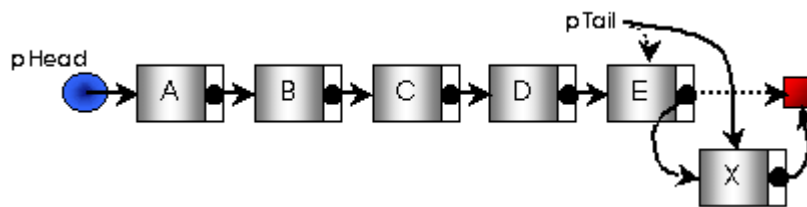
Bước 2.2 : Head = new_ele ;

Cài đặt:

```

void AddFirst(List &l, Node * new_ele)
{
    if (l.pHead==NULL) //Danh sách rỗng
    {
        l.pHead = new_ele;
        l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;
        l.pHead = new_ele;
    }
}

```



Chèn vào cuối danh sách

Thuật toán:

Bắt đầu:

Nếu Danh sách rỗng Thì

Bước 1.1 : Head = new_elelment;

Bước 1.2 : Tail = Head;

Ngược lại

Bước 2.1 : Tail ->pNext = new_ele;

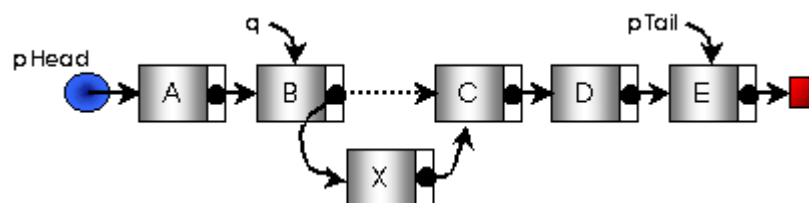
Bước 2.2 : Tail = new_ele ;

Cài đặt:

```

void AddTail(List &l, Node *new_ele)
{
    if (l.pHead==NULL)
    {
        l.pHead = new_ele;
        l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele;
        l.pTail = new_ele;
    }
}

```



Chèn vào danh sách sau một phần tử q

Thuật toán:

Bắt đầu:

Nếu (q != NULL) thì

Bước 1 : new_ele -> pNext = q->pNext;

Bước 2 : q->pNext = new_ele ;

Cài đặt:

```
void AddAfter(List &l, Node *q, Node * new_ele)
{
    if ( q!=NULL)
    {
        new_ele->pNext = q->pNext;
        q->pNext = new_ele;
        if(q == l.pTail)
            l.pTail = new_ele;
    }
    else //chèn vào đầu danh sách
        AddFirst(l, new_ele);
}
```

3.2.2.2. Tìm một phần tử trong danh sách đơn

Thuật toán:

Danh sách đơn đòi hỏi truy xuất tuần tự, do đó chỉ có thể áp dụng thuật toán tìm tuyến tính để xác định phần tử trong danh sách có khoá k. Sử dụng một con trỏ phụ trợ p để lần lượt trở đến các phần tử trong danh sách. Thuật toán được thể hiện như sau:

Bước 1:

p = Head; //Cho p trở đến phần tử đầu danh sách

Bước 2:

Trong khi (p != NULL) và (p->pNext != k) thực hiện:

Bước 2.1 : p:=p->Next; // Cho p trở tới phần tử kế

Bước 3:

Nếu p != NULL thì p trở tới phần tử cần tìm

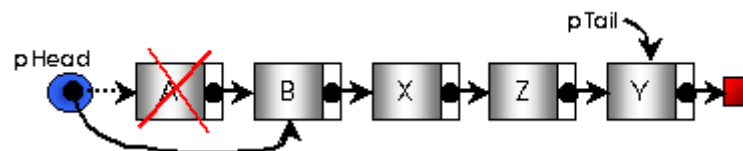
Ngược lại: không có phần tử cần tìm.

Cài đặt:

```
Node *Search(List l, Data k)
{
    Node *p;
    p = l.pHead;
    while( (p!= NULL) && (p->Info != k) )
        p = p->pNext;
    return p;
}
```

3.2.2.3. Hủy một phần tử khỏi danh sách

Có 3 loại thao tác thông dụng hủy một phần tử ra khỏi danh sách. Chúng ta sẽ lần lượt khảo sát chúng. Lưu ý là khi cấp phát bộ nhớ, chúng ta đã dùng hàm new. Vì vậy khi giải phóng bộ nhớ ta phải dùng hàm delete.



Hủy phần tử đầu danh sách:

Thuật toán:

Bắt đầu:

Nếu (Head != NULL) thì

Bước 1: p = Head; // p là phần tử cần hủy

Bước 2:

Bước 2.1 : Head = Head->pNext; // tách p ra khỏi danh sách

Bước 2.2 : free(p); // Hủy biến động do p trở đến

Bước 3:

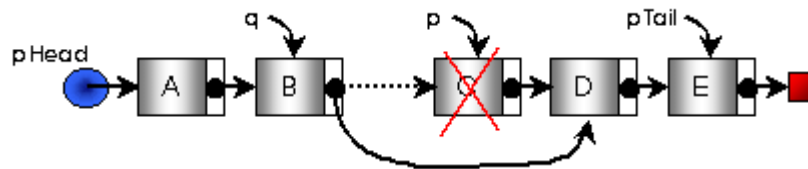
Nếu Head=NULL thì Tail = NULL; //Danh sách rỗng

Cài đặt:

```
Data RemoveHead(List &l)
```

```
{
    Node *p;
    Data x;

    if ( l.pHead != NULL)
    {
        p = l.pHead; x = p->data;
        l.pHead = l.pHead->pNext;
        delete p;
        if(l.pHead == NULL) l.pTail = NULL;
    }
    return x;
}
```



Hủy một phần tử đứng sau phần tử q

Thuật toán:

Bắt đầu:

Nếu (q!= NULL) thì

Bước 1: p = q->Next; // p là phần tử cần hủy

Bước 2: Nếu (p != NULL) thì// q không phải là cuối danh sách

Bước 21: q->Next = p->Next; // tách p ra khỏi danh sách

Bước 22: free(p); // Hủy biến động do p trở đến

Cài đặt:

```
void RemoveAfter (List &l, Node *q)
```

```
{
    NODE *p;
    if ( q != NULL)
    {
        p = q ->pNext ;
        if ( p != NULL)
        {
            if(p == l.pTail)
                l.pTail = q;
            q->pNext = p->pNext;
            delete p;
        }
    }
}
```

```

}
else
RemoveHead(l);
}

```

Hủy 1 phần tử có khoá k

Thuật toán:

Bước 1:

 Tìm phần tử p có khóa k và phần tử q đứng trước nó

Bước 2:

 Nếu (p!= NULL) thì // tìm thấy k

 Hủy p ra khỏi danh sách tương tự hủy phần tử sau q;

 Ngược lại

 Báo không có k;

Cài đặt:

```

int RemoveNode(List &l, Data k)
{
    Node *p = l.pHead;
    Node *q = NULL;
    while( p != NULL)
    {
        if(p->Info == k) break;
        q = p; p = p->pNext;
    }
    if(p == NULL) return 0; //Không tìm thấy k
    if(q != NULL)
    {
        if(p == l.pTail)
            l.pTail = q;
        q->pNext = p->pNext;
        delete p;
    }
    else //p là phần tử đầu danh sách
    {
        l.pHead = p->pNext;
        if(l.pHead == NULL)
            l.pTail = NULL;
    }
    return 1;
}

```

3.2.2.4. Duyệt danh sách

Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách như:

- Đếm các phần tử của danh sách,
- Tìm tất cả các phần tử thoả điều kiện,
- Hủy toàn bộ danh sách (và giải phóng bộ nhớ)
- Để duyệt danh sách (và xử lý từng phần tử) ta thực hiện các thao tác sau:

Thuật toán:

Bước 1:

 p = Head; //Cho p trở đến phần tử đầu danh sách

Bước 2:

Trong khi (Danh sách chưa hết) thực hiện

Bước 2.1 : Xử lý phần tử p;

Bước 2.2 : $p = p \rightarrow \text{pNext}$; // Cho p trở tới phần tử kế

Cài đặt:

```
void ProcessList (List &l)
{
    Node *p;
    p = l.pHead;
    while (p!= NULL)
    {
        ProcessNode(p); // xử lý cụ thể tùy ứng dụng
        p = p->pNext;
    }
}
```

Lưu ý:

Để huỷ toàn bộ danh sách, ta có một chút thay đổi trong thủ tục duyệt (xử lý) danh sách trên (ở đây, thao tác xử lý bao gồm hành động giải phóng một phần tử, do vậy phải cập nhật các liên kết liên quan):

Thuật toán:

Bước 1:

Trong khi (Danh sách chưa hết) thực hiện

Bước 1.1:

$p = \text{Head}$;

$\text{Head} = \text{Head} \rightarrow \text{pNext}$; // Cho p trở tới phần tử kế

Bước 1.2:

Hủy p;

Bước 2:

$\text{Tail} = \text{NULL}$; //Bảo đảm tính nhất quán khi danh sách rỗng

Cài đặt:

```
void ReamoveList (List &l)
{
    Node *p;
    while (l.pHead!= NULL)
    {
        p = l.pHead;
        l.pHead = p->pNext;
        delete p;
    }
    l.pTail = NULL;
}
```

3.3. DANH SÁCH LIÊN KẾT KÉP

3.3.1. Tổ chức danh sách liên kết kép

Danh sách liên kết kép là danh sách mà mỗi phần tử trong danh sách có kết nối với 1 phần tử đứng trước và 1 phần tử đứng sau nó.



Các khai báo sau định nghĩa một danh sách liên kết kép đơn giản trong đó ta dùng hai con trỏ: pPrev liên kết với phần tử đứng trước và pNext như thường lệ, liên kết với phần tử đứng sau:

```

typedef      struct   DNode
{
    Data Info;
    DNode* pPre;    // trỏ đến phần tử đứng trước
    DNode* pNext;    // trỏ đến phần tử đứng sau
}DNode;
typedef      struct   DList
{
    DNode* pHead;    // trỏ đến phần tử đầu danh sách
    DNode* pTail;    // trỏ đến phần tử cuối danh sách
}DList;

```

Khi đó, thủ tục khởi tạo một phần tử cho danh sách liên kết kép được viết lại như sau :

```

DNode* GetNode (Data x)
{
    DNode *p;
    // Cấp phát vùng nhớ cho phần tử
    p = new DNode;
    if ( p==NULL)
    {
        printf("Không đủ bộ nhớ");
        exit(1);
    }
    // Gán thông tin cho phần tử p
    p ->Info = x;
    p->pPrev = NULL;
    p->pNext = NULL;
    return p;
}

```

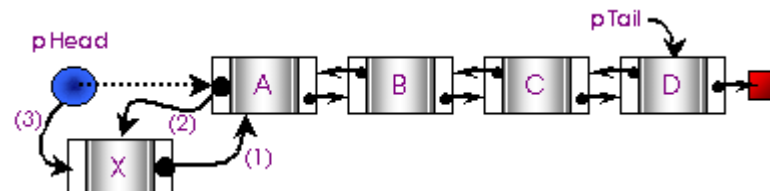
Tương tự danh sách liên kết đơn, ta có thể xây dựng các thao tác cơ bản trên danh sách liên kết kép (danh sách kép). Một số thao tác không khác gì trên danh sách đơn. Dưới đây là một số thao tác đặc trưng của danh sách kép:

3.3.2. Các thao tác cơ bản trên danh sách kép

3.3.2.1. Chèn một phần tử vào danh sách:

Có 4 loại thao tác chèn new_ele vào danh sách:

Chèn vào đầu danh sách



Thuật toán:

Nếu Danh sách rỗng Thì

Bước 1.1 : pHead = new_ele;

Bước 1.2 : pTail = pHead;

Ngược lại

Bước 2.1 : new_ele ->pNext = pHead;

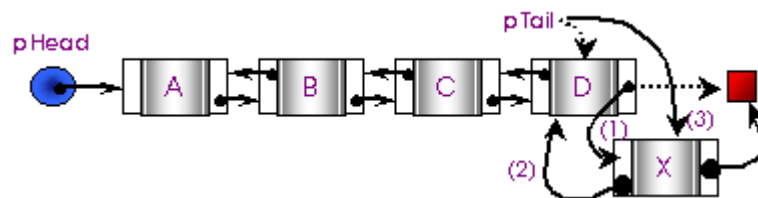
Bước 2.2 : pHead ->pPrev = new_ele;

Bước 2.3 : pHead = new_ele ;

Cài đặt:

```
void AddFirst(DList &l, DNode* new_ele)
{
    if (l.pHead==NULL) //Danh sách rỗng
    {
        l.pHead = new_ele;
        l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;    // (1)
        l.pHead ->pPrev = new_ele;    // (2)
        l.pHead = new_ele;    // (3)
    }
}
```

```
NODE* InsertHead(DList &l, Data x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele ==NULL) return NULL;
    if (l.pHead==NULL)
    {
        l.pHead = new_ele;
        l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;    // (1)
        l.pHead ->pPrev = new_ele;    // (2)
        l.pHead = new_ele;    // (3)
    }
    return new_ele;
}
```

Chèn vào cuối danh sách**Thuật toán:**

Nếu Danh sách rỗng Thì

Bước 1.1 : pHead = new_ele;

Bước 1.2 : pTail = pHead;

Ngược lại

Bước 2.1: pTail->Next = new_ele;

Bước 2.2: new_ele ->pPrev =pTail;

Bước 2.3: pTail = new_ele;

Cài đặt:

```
void AddTail(DList &l, DNode *new_ele)
```

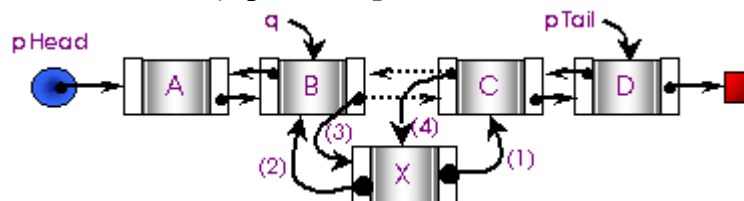


```

{
if (l.pHead==NULL)
{
    l.pHead = new_ele;
    l.pTail = l.pHead;
}
else
{
    l.pTail->Next = new_ele; // (1)
    new_ele ->pPrev = l.pTail; // (2)
    l.pTail = new_ele; // (3)
}
}
}
NODE* InsertTail(DList &l, Data x)
{
    NODE* new_ele = GetNode(x);
    if (new_ele ==NULL) return NULL;
    if (l.pHead==NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele; // (1)
        new_ele ->pPrev = l.pTail; // (2)
        l.pTail = new_ele; // (3)
    }
    return new_ele;
}

```

Chèn vào danh sách sau một phần tử q



Thuật toán:

Bước 1 : $p = q \rightarrow pNext$

Nếu $q \neq \text{NULL}$ thì

Bước 2.1: $\text{new_ele} \rightarrow pNext = p$;

Bước 2.2: $\text{new_ele} \rightarrow pPrev = q$;

Bước 2.3: $q \rightarrow pNext = \text{new_ele}$;

Nếu $p \neq \text{NULL}$ thì

Bước 2.4: $p \rightarrow pPrev = \text{new_ele}$;

Nếu $q == l.pTail$ thì

Bước 2.5: $l.pTail = \text{new_ele}$;

Ngược lại

$\text{AddFirst}(l, \text{new_ele}); // \text{Chèn vào đầu danh sách}$

Cài đặt:

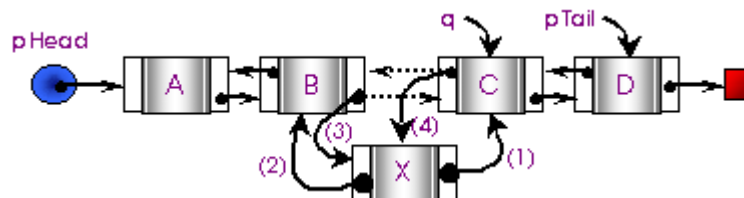
```

void AddAfter(DList &l, DNode* q, DNode* new_ele)
{
    DNode* p = q->pNext;
    if ( q!=NULL)
    {
        new_ele->pNext = p; //(1)
        new_ele->pPrev = q; //(2)
        q->pNext = new_ele; //(3)
        if(p != NULL)
            p->pPrev = new_ele; //(4)
        if(q == l.pTail)
            l.pTail = new_ele;
    }
    else //chèn vào đầu danh sách
        AddFirst(l, new_ele);
}

void InsertAfter(DList &l, DNode *q, Data x)
{
    DNode* p = q->pNext;
    NODE* new_ele = GetNode(x);
    if (new_ele ==NULL) return NULL;
    if ( q!=NULL)
    {
        new_ele->pNext = p; //(1)
        new_ele->pPrev = q; //(2)
        q->pNext = new_ele; //(3)
        if(p != NULL)
            p->pPrev = new_ele; //(4)
        if(q == l.pTail)
            l.pTail = new_ele;
    }
    else //chèn vào đầu danh sách
        AddFirst(l, new_ele);
}

```

Chèn vào danh sách trước một phần tử q



Thuật toán:

Bước 1 : p = q->pPrev;
 Nếu q!= NULL thì
 Bước 2.1: new_ele->pNext = q;
 Bước 2.2: new_ele->pPrev = p;
 Bước 2.3: q->pPrev = new_ele;
 Nếu p != NULL thì
 Bước 2.4: p->pNext = new_ele;

Nếu $q == pHead$ thì
 Bước 2.5: $pHead = new_ele$;

Ngược lại

$AddTail(l, new_ele)$; //Chèn vào cuối danh sách

Cài đặt:

```
void AddBefore(DList &l, DNode q, DNode * new_ele)
{
    DNode * p = q->pPrev;
    if ( q!=NULL)
    {
        new_ele->pNext = q; //(1)
        new_ele->pPrev = p; //(2)
        q->pPrev = new_ele; //(3)
        if(p != NULL)
            p->pNext = new_ele; //(4)
        if(q == l.pHead)
            l.pHead = new_ele;
    }
    else //chèn vào đầu danh sách
        AddTail(l, new_ele);
}
```

```
void InsertBefore(DList &l, Dnode * q, Data x)
{
    DNode * p = q->pPrev;
    NODE* new_ele = GetNode(x);
    if (new_ele ==NULL) return NULL;
    if ( q!=NULL)
    {
        new_ele->pNext = q; //(1)
        new_ele->pPrev = p; //(2)
        q->pPrev = new_ele; //(3)
        if(p != NULL)
            p->pNext = new_ele; //(4)
        if(q == l.pHead)
            l.pHead = new_ele;
    }
    else //chèn vào đầu danh sách
        AddTail(l, new_ele);
}
```

3.3.2.2. Hủy một phần tử khỏi danh sách

Có 5 loại thao tác thông dụng hủy một phần tử ra khỏi danh sách. Chúng ta sẽ lần lượt khảo sát chúng.

Hủy phần tử đầu danh sách:

Thuật toán:

Bắt đầu:

 Nếu ($pHead \neq NULL$) thì
 Bước 1: $p = pHead$;
 Bước 2: Lưu giữ dữ liệu của phần tử p
 Bước 3: $pHead = l.pHead->pNext$;
 Bước 4: $pHead->pPrev = NULL$;

Bước 5: Xóa phần tử p// delete (p);
 Nếu pHead == NULL thì
 pTail = NULL;
 Ngược lại
 pHead->pPrev = NULL;
 Bước 6: Trả giá trị dữ liệu của p;

Cài đặt:

```
Data RemoveHead(DList &l)
{
    DNode    *p;
    Data x;
    if ( l.pHead != NULL)
    {
        p = l.pHead;
        x = p->data;
        l.pHead = l.pHead->pNext;
        l.pHead->pPrev = NULL;
        delete p;
        if(l.pHead == NULL)
            l.pTail = NULL;
        else
            l.pHead->pPrev = NULL;
    }
    return x;
}
```

Hủy phần tử cuối danh sách:

Thuật toán:

Nếu pTail != NULL thì
 Bước 1: p = pTail; // p là phần tử cần xóa
 Bước 2: x = p->data; // lưu dữ liệu của p
 Bước 3 : pTail = pTail->pPrev;
 Bước 3: pTail->pNext = NULL;
 Bước 5: delete p;
 Nếu pHead == NULL thì
 pTail = NULL;
 Ngược lại
 pHead->pPrev = NULL;
 Bước 6: Trả giá dữ liệu của phần tử p;

Cài đặt:

```
Data RemoveTail(DList &l)
{
    DNode *p;
    Data x;
    if ( l.pTail != NULL)
    {
        p = l.pTail; x = p->data;
        l.pTail = l.pTail->pPrev;
        l.pTail->pNext = NULL;
        delete p;
        if(l.pHead == NULL)
```

```

        l.pTail = NULL;
    else
        l.pHead->pPrev = NULL;
    }
    return x;
}

```

Hủy một phần tử đứng sau phần tử q

Thuật toán:

Nếu pTail != NULL thì
 Bước 1: p = q->pNext; // p là phần tử cần xóa
 Nếu p != NULL thì
 Bước 2:
 q->pNext = p->pNext;
 Nếu p == l.pTail thì
 pTail = q;
 Ngược lại
 p->pNext->pPrev = q;
 delete p;
 Ngược lại
 Bước 3: RemoveHead(l);

Cài đặt:

```

void RemoveAfter (DList &l, DNode *q)
{
    DNode *p;
    if ( q != NULL)
    {
        p = q ->pNext ;
        if ( p != NULL)
        {
            q->pNext = p->pNext;
            if(p == l.pTail)
                l.pTail = q;
            else
                p->pNext->pPrev = q;
            delete p;
        }
    }
    else
        RemoveHead(l) ;
}

```

Hủy một phần tử đứng trước phần tử q

Thuật toán:

Nếu pTail != NULL thì
 Bước 1: p = q->pPrev; // p là phần tử cần xóa
 Nếu p != NULL thì
 Bước 2:
 q->pPrev = p->pPrev;
 Nếu p == l.pHead thì
 pHead = q;
 Ngược lại
 p->pPrev->pNext = q;

```

        delete p;
    Ngược lại
        Bước 3: RemoveTail(l);
Cài đặt:

void RemoveAfter (DList &l, DNode *q)
{
    DNode      *p;
    if ( q != NULL)
    {
        p = q ->pPrev;
        if ( p != NULL)
        {
            q->pPrev = p->pPrev;
            if(p == l.pHead)
                l.pHead = q;
            else
                p->pPrev->pNext = q;
            delete p; }
        }
    else
        RemoveTail(l);
}

```

Hủy 1 phần tử có khoá k

Thuật toán:

```

    Bước 1: Tìm khóa k
    Nếu tìm thấy thì
        Bước 1: q = p->pPrev; // p là phần tử cần xóa
        Nếu q!= NULL thì
            Bước 2:
                p=q->pNext;
                Nếu p != NULL thì
                    q->pNext = p->pNext;
                Nếu p == pTail thì
                    pTail = q;
                Ngược lại
                    p->pNext->pPrev = q;
            Ngược lại //p là phần tử đầu danh sách
        Bước 3:
            pHead = p->pNext;
            Nếu pHead == NULL thì
                pTail = NULL;
            Ngược lại
                pHead->pPrev = NULL;
        Bước 4:
            delete p;

```

Cài đặt:

```

int RemoveNode(DList &l, Data k)
{

```

```

DNode      *p = l.pHead;
NODE *q;
while( p != NULL)
{
    if(p->Info == k) break;
    p = p->pNext;
}
if(p == NULL) return 0; //Không tìm thấy k
q = p->pPrev;
if ( q != NULL)
{
    p = q ->pNext ;
    if ( p != NULL)
    {
        q->pNext = p->pNext;
        if(p == l.pTail)
            l.pTail = q;
        else
            p->pNext->pPrev = q;
    }
}
else //p là phần tử đầu danh sách
{
    l.pHead = p->pNext;
    if(l.pHead == NULL)
        l.pTail = NULL;
    else
        l.pHead->pPrev = NULL;
}
delete p;
return 1;
}

```

Nhận xét:

Danh sách kép về mặt cơ bản có tính chất giống như danh sách đơn. Tuy nhiên nó có một số tính chất khác danh sách đơn như sau:

Danh sách kép có mỗi liên kết hai chiều nên từ một phần tử bất kỳ có thể truy xuất một phần tử bất kỳ khác. Trong khi trên danh sách đơn ta chỉ có thể truy xuất đến các phần tử đứng sau một phần tử cho trước. Điều này dẫn đến việc ta có thể dễ dàng hủy phần tử cuối danh sách kép, còn trên danh sách đơn thao tác này tốn chi phí $O(n)$.

Bù lại, danh sách kép tốn chi phí gấp đôi so với danh sách đơn cho việc lưu trữ các mối liên kết. Điều này khiến việc cập nhật cũng nặng nề hơn trong một số trường hợp. Như vậy ta cần cân nhắc lựa chọn CTDL hợp lý khi cài đặt cho một ứng dụng cụ thể.

❖ Bài tập củng cố:

1. Phân tích ưu, khuyết điểm của cấu trúc liên kết so với mảng. Tổng quát hóa các trường hợp nên dùng cấu trúc liên kết.

2. Xây dựng một cấu trúc dữ liệu thích hợp để biểu diễn đa thức $P(x)$ có dạng:

$$P(x) = c_1x^n + c_2x^{n^2} + \dots + c_kx^{n_k}$$

Biết rằng:

Các thao tác xử lý trên đa thức bao gồm:

- Thêm một phần tử vào cuối đa thức
- In danh sách các phần tử trong đa thức theo:
 - thứ tự nhập vào
 - ngược với thứ tự nhập vào
- Hủy một phần tử bất kỳ trong danh sách

Số lượng các phần tử không hạn chế

Chỉ có nhu cầu xử lý đa thức trong bộ nhớ chính.

- Giải thích lý do chọn CTDL đã định nghĩa.
- Viết chương trình con ước lượng giá trị của đa thức $P(x)$ khi biết x .
- Viết chương trình con rút gọn biểu thức (gộp các phần tử cùng số mũ).

3. Xét đoạn chương trình tạo một cấu trúc đơn gồm 4 phần tử (không quan tâm dữ liệu) sau đây:

```
Dx = NULL; p=Dx;
Dx = new (NODE);
for(i=0; i < 4; i++)
{
    p = p->next;
    p = new (NODE);
}
p->next = NULL;
```

Đoạn chương trình có thực hiện được thao tác tạo nêu trên không? Tại sao? Nếu không thì có thể sửa lại như thế nào cho đúng?

4. Một ma trận chỉ chứa rất ít phần tử với giá trị có nghĩa (ví dụ: phần tử $\neq 0$) được gọi là ma trận thưa.

Ví dụ :

$$\begin{pmatrix} 0 & 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \end{pmatrix}$$

Dùng cấu trúc cấu trúc liên kết để tổ chức biểu diễn một ma trận thưa sao cho tiết kiệm nhất (chỉ lưu trữ các phần tử có nghĩa).

- Viết chương trình cho phép nhập, xuất ma trận.
- Viết chương trình con cho phép cộng hai ma trận.

CHƯƠNG 4

CẤU TRÚC DỮ LIỆU NGĂN XẾP (STACK) VÀ HÀNG ĐỢI (QUEUE)

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể:

- Vận dụng và cài đặt được các phép toán trên ngăn xếp.
- Vận dụng và cài đặt được các phép toán trên hàng đợi.

4.1. NGĂN XẾP (STACK)

Stack là một vật chứa (container) các đối tượng làm việc theo cơ chế LIFO (Last In First Out) nghĩa là việc thêm một đối tượng vào stack hoặc lấy một đối tượng ra khỏi stack được thực hiện theo cơ chế "Vào sau ra trước".

Các đối tượng có thể được thêm vào stack bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào sau cùng mới được phép lấy ra khỏi stack.

Thao tác thêm 1 đối tượng vào stack thường được gọi là "Push". Thao tác lấy 1 đối tượng ra khỏi stack gọi là "Pop".

Trong tin học, CTDL stack có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều sâu và quay lui, vết cặn, ứng dụng trong các bài toán tính toán biểu thức, .



Một hình ảnh một stack

Ta có thể định nghĩa CTDL stack như sau: stack là một CTDL trừu tượng (ADT) tuyến tính hỗ trợ 2 thao tác chính:

Push(o): Thêm đối tượng o vào đầu stack

Pop(): Lấy đối tượng ở đầu stack ra khỏi stack và trả về giá trị của nó. Nếu stack rỗng thì lỗi sẽ xảy ra.

Ngoài ra, stack cũng hỗ trợ một số thao tác khác:

isEmpty(): Kiểm tra xem stack có rỗng không.

Top(): Trả về giá trị của phần tử nằm ở đầu stack mà không hủy nó khỏi stack. Nếu stack rỗng thì lỗi sẽ xảy ra.

Các thao tác thêm, trích và hủy một phần tử chỉ được thực hiện ở cùng một phía của Stack do đó hoạt động của Stack được thực hiện theo nguyên tắc LIFO (Last In First Out - vào sau ra trước).

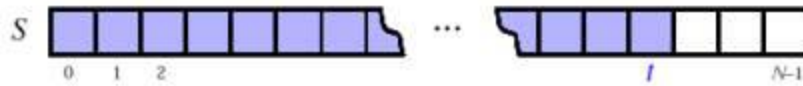
Để biểu diễn Stack, ta có thể dùng mảng 1 chiều hoặc dùng danh sách liên kết.

4.1.1. Biểu diễn Stack dùng mảng:

Ta có thể tạo một stack bằng cách khai báo một mảng 1 chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000).

Như vậy stack có thể chứa tối đa N phần tử đánh số từ 0 đến N - 1. Phần tử nằm ở đầu stack sẽ có chỉ số t (lúc đó trong stack đang chứa t+1 phần tử)

Để khai báo một stack, ta cần một mảng 1 chiều S, biến nguyên t cho biết chỉ số của đầu stack và hằng số N cho biết kích thước tối đa của stack.



Tạo stack S và quản lý đỉnh stack bằng biến t:

Data S [N];

Int t;

Việc cài đặt stack thông qua mảng một chiều đơn giản và khá hiệu quả.

Tuy nhiên, hạn chế lớn nhất của phương án cài đặt này là giới hạn về kích thước của stack N. Giá trị của N có thể quá nhỏ so với nhu cầu thực tế hoặc quá lớn sẽ làm lãng phí bộ nhớ.

4.1. 2. Biểu diễn Stack dùng danh sách:

Ta có thể tạo một stack bằng cách sử dụng một danh sách liên kết đơn (danh sách liên kết). Có thể nói, danh sách liên kết có những đặc tính rất phù hợp để dùng làm stack vì mọi thao tác trên stack đều diễn ra ở đầu stack.

Sau đây là các thao tác tương ứng cho list-stack:

Tạo Stack S rỗng

```
LIST * S;
```

Lệnh S.pHead=l.pTail= NULL tạo ra một Stack S rỗng.

Kiểm tra stack rỗng :

```
char IsEmpty(LIST &S)
{
    if (S.pHead == NULL) // stack rỗng
        return 1;
    else return 0;
}
```

Thêm một phần tử p vào stack S

```
void Push(LIST &S, Data x)
{
    InsertHead(S, x);
}
```

Trích huỷ phần tử ở đỉnh stack S

```
Data Pop(LIST &S)
{ Data x;
  if(IsEmpty(S)) return NULLDATA;
  x = RemoveFirst(S);
  return x;
}
```

Xem thông tin của phần tử ở đỉnh stack S

```
Data Top(LIST &S)
{ if(IsEmpty(S)) return NULLDATA;
  return l.Head->Info;
}
```

4.1. 3. Ứng dụng của Stack

- Cấu trúc Stack thích hợp lưu trữ các loại dữ liệu mà trình tự truy xuất ngược với trình tự lưu trữ, do vậy một số ứng dụng sau thường cần đến stack :

- Trong trình biên dịch (thông dịch), khi thực hiện các thủ tục, Stack được sử dụng để lưu môi trường của các thủ tục.

- Trong một số bài toán của lý thuyết đồ thị (như tìm đường đi), Stack cũng thường được sử dụng để lưu dữ liệu khi giải các bài toán này.

Ngoài ra, Stack cũng còn được sử dụng trong trường hợp khử đệ qui đuôi.

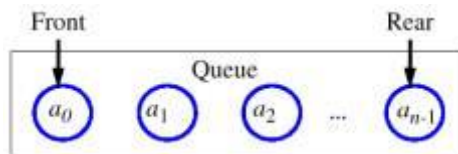
4.2. HÀNG ĐỢI (QUEUE)

Hàng đợi là một vật chứa (container) các đối tượng làm việc theo cơ chế FIFO (First In First Out) nghĩa là việc thêm một đối tượng vào hàng đợi hoặc lấy một đối tượng ra khỏi hàng đợi được thực hiện theo cơ chế "Vào trước ra trước".

Các đối tượng có thể được thêm vào hàng đợi bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào đầu tiên mới được phép lấy ra khỏi hàng đợi.

Thao tác thêm một đối tượng vào hàng đợi và lấy một đối tượng ra khỏi hàng đợi lần lượt được gọi là "enqueue" và "dequeue".

Việc thêm một đối tượng vào hàng đợi luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi.



Trong tin học, CTDL hàng đợi có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng và quay lui, vết cạn, tổ chức quản lý và phân phối tiến trình trong các hệ điều hành, tổ chức bộ đệm bàn phím, .

Ta có thể định nghĩa CTDL hàng đợi như sau: hàng đợi là một CTDL trừu tượng (ADT) tuyến tính. Tương tự như stack, hàng đợi hỗ trợ các thao tác:

EnQueue(o): Thêm đối tượng o vào cuối hàng đợi

DeQueue(): Lấy đối tượng ở đầu queue ra khỏi hàng đợi và trả về giá trị của nó.

Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.

IsEmpty(): Kiểm tra xem hàng đợi có rỗng không.

Front(): Trả về giá trị của phần tử nằm ở đầu hàng đợi mà không hủy nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.

Các thao tác thêm, trích và hủy một phần tử phải được thực hiện ở 2 phía khác nhau của hàng đợi do đó hoạt động của hàng đợi được thực hiện theo nguyên tắc FIFO (First In First Out - vào trước ra trước).

Cũng như stack, ta có thể dùng cấu trúc mảng 1 chiều hoặc cấu trúc danh sách liên kết để biểu diễn cấu trúc hàng đợi.

4.2.1. Biểu diễn dùng mảng:

Ta có thể tạo một hàng đợi bằng cách sử dụng một mảng 1 chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000) theo kiểu xoay vòng (coi phần tử a_{n-1} kề với phần tử a_0).

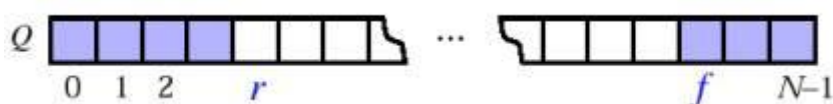
Như vậy hàng đợi có thể chứa tối đa N phần tử. Phần tử nằm ở đầu hàng đợi (front element) sẽ có chỉ số f. Phần tử nằm ở cuối hàng đợi (rear element) sẽ có chỉ số r (xem hình).

Để khai báo một hàng đợi, ta cần một mảng một chiều Q, hai biến nguyên f, r cho biết chỉ số của đầu và cuối của hàng đợi và hằng số N cho biết kích thước tối đa của hàng đợi. Ngoài ra, khi dùng mảng biểu diễn hàng đợi, ta cũng cần một giá trị đặc biệt để gán cho những ô còn trống trên hàng đợi. Giá trị này là một giá trị nằm ngoài miền xác định của dữ liệu lưu trong hàng đợi. Ta ký hiệu nó là NULLDATA như ở những phần trước.

Trạng thái hàng đợi lúc bình thường:



Trạng thái hàng đợi lúc xoay vòng:



Câu hỏi đặt ra: khi giá trị $f=r$ cho ta điều gì? Ta thấy rằng, lúc này hàng đợi chỉ có thể ở

một trong hai trạng thái là rỗng hoặc đầy. Coi như một bài tập các bạn hãy tự suy nghĩ tìm câu trả lời trước khi đọc tiếp để kiểm tra kết quả.

Hàng đợi có thể được khai báo cụ thể như sau:

Data Q[N] ;

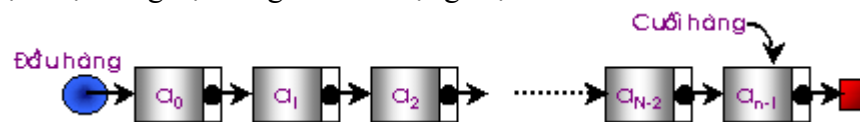
int f, r;

Cũng như stack, do khi cài đặt bằng mảng một chiều, hàng đợi có kích thước tối đa nên ta cần xây dựng thêm một thao tác phụ cho hàng đợi:

IsFull(): Kiểm tra xem hàng đợi có đầy chưa.

4.2.2. Dùng danh sách liên kết:

Ta có thể tạo một hàng đợi bằng cách sử dụng một danh sách liên kết đơn.



Phần tử đầu danh sách liên kết (head) sẽ là phần tử đầu hàng đợi, phần tử cuối danh sách liên kết (tail) sẽ là phần tử cuối hàng đợi.

Sau đây là các thao tác tương ứng cho hàng đợi sử dụng danh sách liên kết đơn:

Tạo hàng đợi rỗng:

Lệnh `Q.pHead = Q.pTail = NULL` tạo ra một hàng đợi rỗng.

Kiểm tra hàng đợi rỗng :

```
char IsEmpty(LIST Q)
{
    if (Q.pHead == NULL) // stack rỗng
        return 1;
    else return 0;
}
```

Thêm một phần tử p vào cuối hàng đợi

```
void EnQueue(LIST Q, Data x)
{
    InsertTail(Q, x);
}
```

Trích/Hủy phần tử ở đầu hàng đợi

```
Data DeQueue(LIST Q)
{
    Data x;
    if (IsEmpty(Q)) return NULLDATA;
    x = RemoveFirst(Q);
    return x;
}
```

Xem thông tin của phần tử ở đầu hàng đợi

```
Data Front(LIST Q)
{
    if (IsEmpty(Q)) return NULLDATA;
    return Q.pHead->Info;
}
```

Các thao tác trên đều làm việc với chi phí $O(1)$.

Lưu ý, nếu không quản lý phần tử cuối danh sách, thao tác dequeue sẽ có độ phức tạp $O(n)$.

4.2.3. Ứng dụng của hàng đợi:

- Hàng đợi có thể được sử dụng trong một số bài toán:
- Bài toán sản xuất và tiêu thụ (ứng dụng trong các hệ điều hành song song).

- Bộ đệm (ví dụ: Nhấn phím -> Bộ đệm -> CPU xử lý).
- Xử lý các lệnh trong máy tính (ứng dụng trong HĐH, trình biên dịch), hàng đợi các tiến trình chờ được xử lý, ...

❖ **Bài tập củng cố:**

1. Hãy cho biết nội dung của stack sau mỗi thao tác trong dãy :

EAS*Y**QUE***ST***I*ON

Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào stack, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong stack in lên màn hình.

Hãy cho biết sau khi hoàn tất chuỗi thao tác, những gì xuất hiện trên màn hình ?

2. Hãy cho biết nội dung của hàng đợi sau mỗi thao tác trong dãy :

EAS*Y**QUE***ST***I*ON

Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào hàng đợi, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong hàng đợi in lên màn hình.

Hãy cho biết sau khi hoàn tất chuỗi thao tác, những gì xuất hiện trên màn hình ?

3. Hãy dùng cấu trúc dữ liệu Stack để viết chương trình theo yêu cầu sau:

- a) Tính giá trị của biểu thức hậu tố (ab+: nghĩa là a+b)
- b) Tính giá trị của biểu thức tiền tố (+ab: nghĩa là a+b)

4. Một bệnh nhân khi đến khám bệnh ở bệnh viện phải khai báo các thông tin sau: họ tên, tuổi, địa chỉ, tình trạng bệnh vào tờ khai bệnh cho cán bộ tiếp nhận. Sau đó cán bộ tiếp nhận sẽ gán một số thứ tự cho bệnh nhân này theo thứ tự đến của họ.

- a) Hãy khai cấu trúc dữ liệu để đưa thông tin các bệnh nhân vào hàng đợi dùng cấu trúc danh sách liên kết.
- b) Viết hàm enqueue() để đưa một bệnh nhân vào hàng đợi theo cấu trúc trên.

CHƯƠNG 5

CÂY VÀ CÂY NHỊ PHÂN

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể:

- Hiểu được định nghĩa, một số khái niệm và ví dụ minh họa về cấu trúc cây.
- Vận dụng và cài đặt được cấu trúc cây, duyệt cây trên cây nhị phân.
- Vận dụng và cài đặt được các phép toán trên danh sách liên kết kép.

5.1. CẤU TRÚC CÂY

Định nghĩa 1: cây là một tập hợp T các phần tử (gọi là nút của cây) trong đó có 1 nút đặc biệt được gọi là gốc, các nút còn lại được chia thành những tập rời nhau T_1, T_2, \dots, T_n theo quan hệ phân cấp trong đó T_i cũng là một cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp $i+1$. Quan hệ này người ta còn gọi là quan hệ cha-con.

Định nghĩa 2: cấu trúc cây với kiểu cơ sở T là một nút cấu trúc rỗng được gọi là cây rỗng (NULL). Một nút mà thông tin chính của nó có kiểu T , nó liên kết với một số hữu hạn các cấu trúc cây khác cũng có kiểu cơ sở T . Các cấu trúc này được gọi là những cây con của cây đang xét.

5.1.1. Một số khái niệm cơ bản

Bậc của một nút: là số cây con của nút đó.

Bậc của một cây: là bậc lớn nhất của các nút trong cây (số cây con tối đa của một nút thuộc cây). Cây có bậc n thì gọi là cây n -phân.

Nút gốc: là nút không có nút cha.

Nút lá: là nút có bậc bằng 0.

Nút nhánh: là nút có bậc khác 0 và không phải là gốc.

Mức của một nút:

Mức (gốc (T)) = 0.

Gọi $T_1, T_2, T_3, \dots, T_n$ là các cây con của T_0

Mức $(T_1) = \text{Mức}(T_2) = \dots = \text{Mức}(T_n) = \text{Mức}(T_0) + 1$.

Độ dài đường đi từ gốc đến nút x : là số nhánh cần đi qua kể từ gốc đến x .

Độ dài đường đi tổng của cây:

$$P_T = \sum_{x \in T} P_x$$

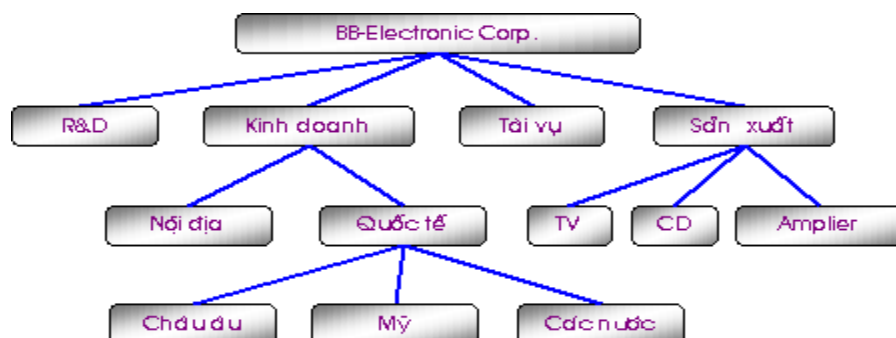
trong đó P_x là độ dài đường đi từ gốc đến x .

Độ dài đường đi trung bình: $PI = P_T/n$ (n là số nút trên cây T).

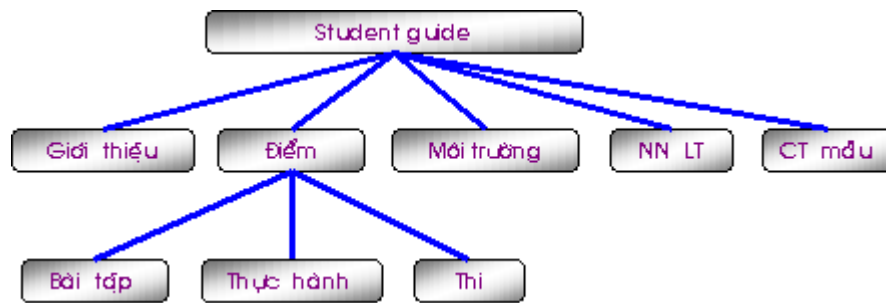
Rừng cây: là tập hợp nhiều cây trong đó thứ tự các cây là quan trọng.

5.1.2. Một số ví dụ cấu trúc cây

Sơ đồ tổ chức của một công ty



Mục lục một quyển sách
Cấu trúc cây thư mục trong DOS/WIN
Cấu trúc thư viện, ...



Nhận xét:

Trong cấu trúc cây không tồn tại chu trình

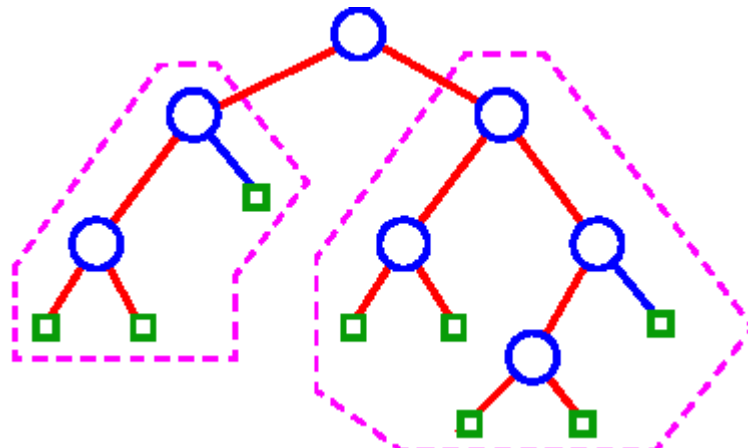
Tổ chức 1 cấu trúc cây cho phép truy cập nhanh đến các phần tử của nó.

5.2. CÂY NHỊ PHÂN

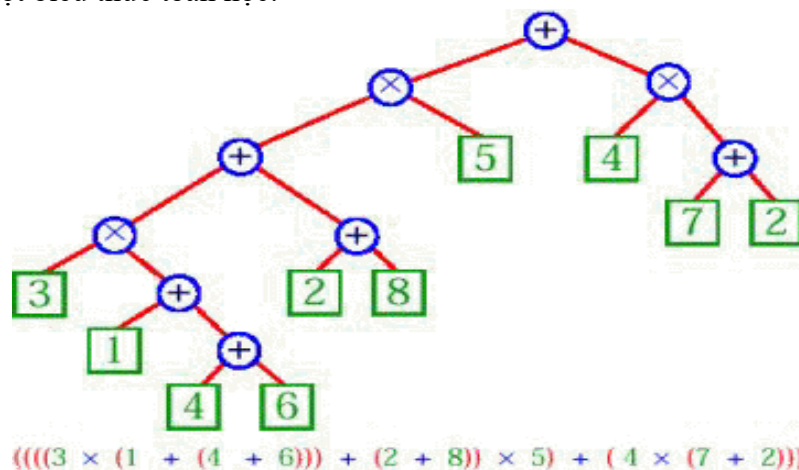
5.2.1. Định nghĩa

Cây nhị phân là cây mà mỗi nút có tối đa 2 cây con.

Trong thực tế thường gặp các cấu trúc có dạng cây nhị phân. Một cây tổng quát có thể biểu diễn thông qua cây nhị phân.



Cây nhị phân có thể ứng dụng trong nhiều bài toán thông dụng. Ví dụ dưới đây cho ta hình ảnh của một biểu thức toán học:



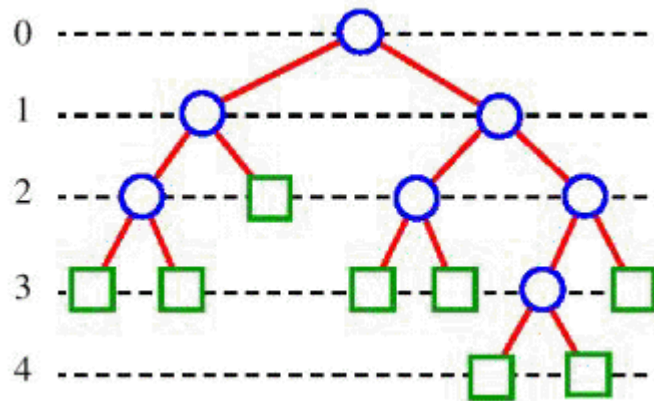
5.2.2. Một số tính chất của cây nhị phân

Số nút nằm ở mức $I \leq 2^I$.

Số nút lá $\leq 2^{h-1}$, với h là chiều cao của cây.

Chiều cao của cây $h \geq \log_2(\text{số nút trong cây})$.

Số nút trong cây $\leq 2^h - 1$.



5.2.3. Biểu diễn cây nhị phân T

Cây nhị phân là một cấu trúc bao gồm các phần tử (nút) được kết nối với nhau theo quan hệ “cha-con” với mỗi cha có tối đa 2 con. Để biểu diễn cây nhị phân ta chọn phương pháp cấp phát liên kết. Ứng với một nút, ta dùng một biến động lưu trữ các thông tin:

Thông tin lưu trữ tại nút.

Địa chỉ nút gốc của cây con trái trong bộ nhớ.

Địa chỉ nút gốc của cây con phải trong bộ nhớ.

Khai báo tương ứng trong ngôn ngữ C có thể như sau:

```
typedef struct    TNODE
{
    Data    Key; //Data là kiểu dữ liệu ứng với thông tin lưu tại nút
    struct NODE *pLeft, *pRight;
} TNODE;
typedef TNODE *TREE;
```

Do tính chất mềm dẻo của cách biểu diễn bằng cấp phát liên kết, phương pháp này được dùng chủ yếu trong biểu diễn cây nhị phân. Từ đây trở đi, khi nói về cây nhị phân, chúng ta sẽ dùng phương pháp biểu diễn này.

5.2.4. Duyệt cây nhị phân

Nếu như khi khảo sát cấu trúc dữ liệu dạng danh sách liên kết ta không quan tâm nhiều đến bài toán duyệt qua tất cả các phần tử của chúng thì bài toán duyệt cây hết sức quan trọng. Nó là cốt lõi của nhiều thao tác quan trọng khác trên cây. Do cây nhị phân là một cấu trúc dữ liệu phi tuyến nên bài toán duyệt cây là bài toán không tầm thường.

Có nhiều kiểu duyệt cây khác nhau, và chúng cũng có những ứng dụng khác nhau. Đối với cây nhị phân, do cấu trúc đệ quy của nó, việc duyệt cây tiếp cận theo kiểu đệ quy là hợp lý và đơn giản nhất. Sau đây chúng ta sẽ xem xét một số kiểu duyệt thông dụng.

Có 3 kiểu duyệt chính có thể áp dụng trên cây nhị phân: duyệt theo thứ tự trước (NLR), thứ tự giữa (LNR) và thứ tự sau (LRN). Tên của 3 kiểu duyệt này được đặt dựa trên trình tự của việc thăm nút gốc so với việc thăm 2 cây con.

Duyệt theo thứ tự trước (Node-Left-Right)

Kiểu duyệt này trước tiên thăm nút gốc sau đó thăm các nút của cây con trái rồi đến cây con phải. Thủ tục duyệt có thể trình bày đơn giản như sau:

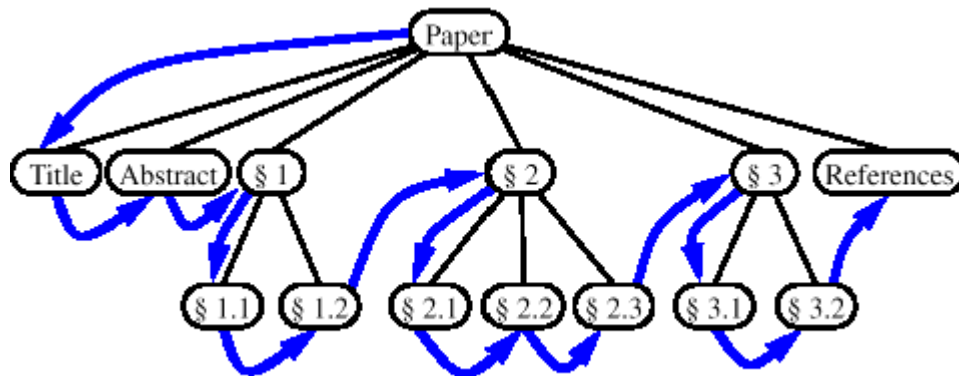
```
void NLR(TREE Root)
```



```

{
if (Root != NULL)
{
    <Xử lý Root>;    //Xử lý tương ứng theo nhu cầu
    NLR(Root->pLeft);
    NLR(Root->pRight);
}
}

```



Duyệt theo thứ tự giữa (Left- Node-Right)

Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm nút gốc rồi đến cây con phải. Thủ tục duyệt có thể trình bày đơn giản như sau:

```

void LNR(TREE Root)
{
if (Root != NULL)
{
    LNR(Root->Left);
    <Xử lý Root>; //Xử lý tương ứng theo nhu cầu
    LNR(Root->Right);
}
}

```

Duyệt theo thứ tự sau (Left-Right-Node)

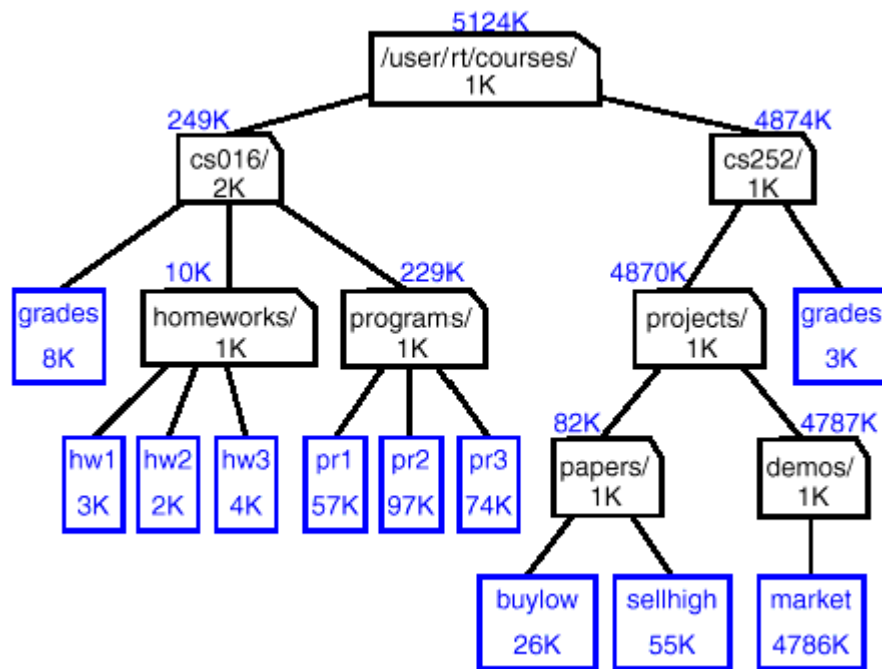
Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm đến cây con phải rồi cuối cùng mới thăm nút gốc. Thủ tục duyệt có thể trình bày đơn giản như sau:

```

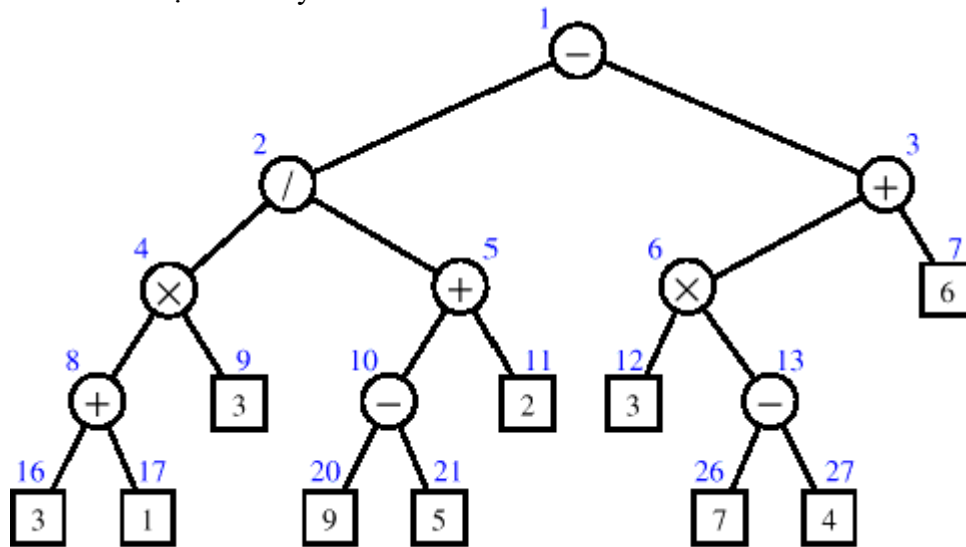
void LRN(TREE Root)
{
if (Root != NULL)
{
    LRN(Root->Left);
    LRN(Root->Right);
    <Xử lý Root>; //Xử lý tương ứng theo nhu cầu
}
}

```

Một ví dụ quen thuộc trong tin học về ứng dụng của duyệt theo thứ tự sau là việc xác định tổng kích thước của một thư mục trên đĩa như hình sau:



Một ứng dụng quan trọng khác của phép duyệt cây theo thứ tự sau là việc tính toán giá trị của biểu thức dựa trên cây biểu thức như hình dưới:



$$(3 + 1) \times 3 / (9 - 5 + 2) - (3 \times (7 - 4) + 6) = -13$$

5.2.5. Biểu diễn cây tổng quát bằng cây nhị phân

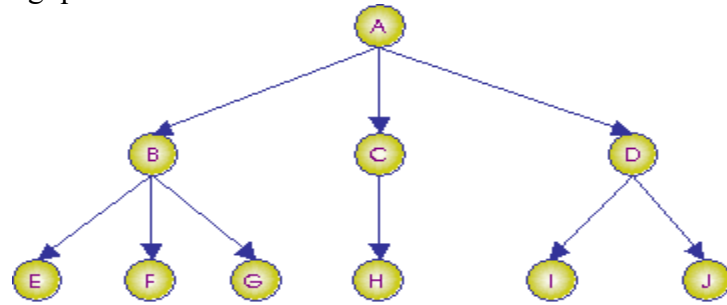
Nhược điểm của các cấu trúc cây tổng quát là bậc của các nút trên cây có thể dao động trong một biên độ lớn \Rightarrow việc biểu diễn gặp nhiều khó khăn và lãng phí. Hơn nữa, việc xây dựng các thao tác trên cây tổng quát phức tạp hơn trên cây nhị phân nhiều. Vì vậy, thường nếu không quá cần thiết phải sử dụng cây tổng quát, người ta chuyển cây tổng quát thành cây nhị phân.

Ta có thể biến đổi một cây bất kỳ thành một cây nhị phân theo qui tắc sau:

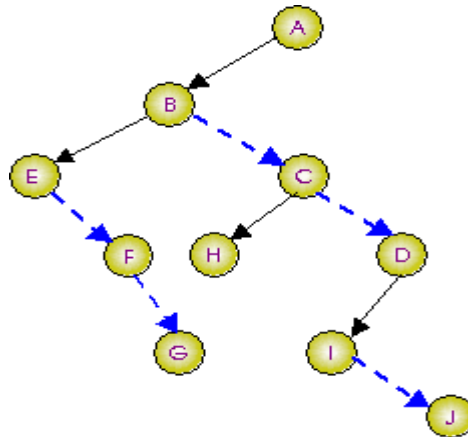
- Giữ lại nút con trái nhất làm nút con trái.
- Các nút con còn lại chuyển thành nút con phải.
- Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu.

Ta có thể xem ví dụ dưới đây để thấy rõ hơn qui trình.

Giả sử có cây tổng quát như hình bên dưới:



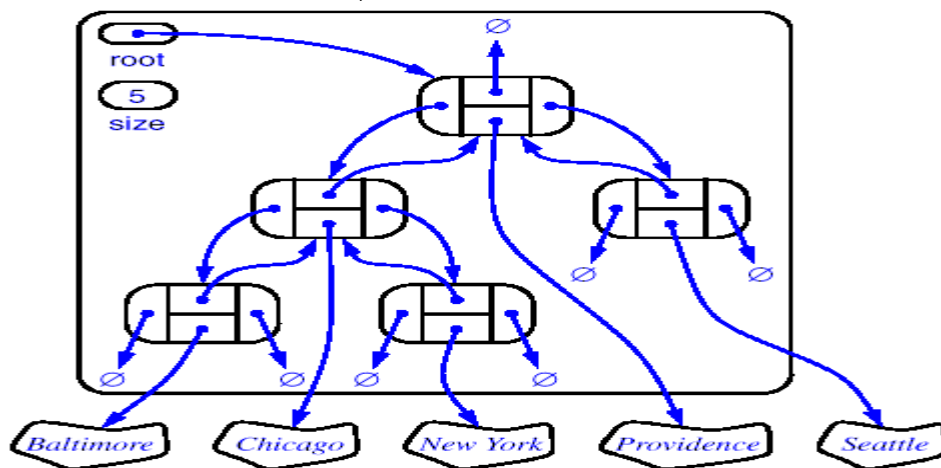
Cây nhị phân tương ứng sẽ như sau:



5.2.6. Một cách biểu diễn cây nhị phân khác

Đôi khi, khi định nghĩa cây nhị phân, người ta quan tâm đến cả quan hệ 2 chiều cha con chứ không chỉ một chiều như định nghĩa ở phần trên. Lúc đó, cấu trúc cây nhị phân có thể định nghĩa lại như sau:

```
typedef struct tagTNode
{
    DataType Key;
    struct tagTNode* pParent;
    struct tagTNode* pLeft;
    struct tagTNode* pRight;
}TNODE;
typedef TNODE    *TREE;
```



❖ **Bài tập củng cố:**

1. Khai báo cấu trúc cây nhị phân (dữ liệu là số nguyên) và viết các hàm sau:
 - a) 3 hàm duyệt cây với xử lý nút là in dữ liệu của nút ra màn hình.
 - b) Đếm tổng số nút trên cây.
 - c) Đếm số nút lá trên cây.
 - d) Tính chiều cao của cây.
2. Viết các hàm xác định các thông tin của cây nhị phân T:
 - a) Số nút lá
 - b) Số nút có đúng 1 cây con
 - c) Số nút có đúng 2 cây con
 - d) Chiều cao của cây
 - e) In ra tất cả các nút ở tầng (mức) thứ k của cây T
 - f) In ra tất cả các nút theo thứ tự từ mức 0 đến mức h-1 của cây T (h là chiều cao của T).
 - g) Độ lệch lớn nhất trên cây. (Độ lệch của một nút là độ lệch giữa chiều cao của cây con trái và cây con phải của nó. Độ lệch lớn nhất trên cây là độ lệch của nút có độ lệch lớn nhất).
3. Viết hàm chuyển một cây nhị phân tìm kiếm thành xâu kép có thứ tự tăng dần.
4. Viết hàm chuyển một cây N-phân thành cây nhị phân.
5. Viết chương trình con đảo nhánh (nhánh trái của một nút trên cây trở thành nhánh phải của nút đó và ngược lại) một cây nhị phân .

CHƯƠNG 6

CÂY NHỊ PHÂN TÌM KIẾM

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể:

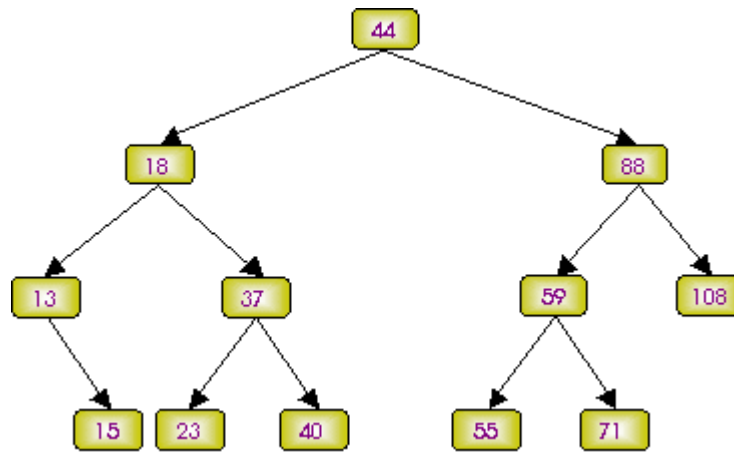
- Hiểu được định nghĩa cấu trúc cây nhị phân tìm kiếm.
- Vận dụng và cài đặt được cấu trúc cây, duyệt cây của cây nhị phân vào cây nhị phân tìm kiếm.
- Vận dụng và cài đặt được các phép toán trên cây nhị phân tìm kiếm.

6.1. CÂY NHỊ PHÂN TÌM KIẾM

Định nghĩa:

Cây nhị phân tìm kiếm (cây nhị phân tìm kiếm) là cây nhị phân trong đó tại mỗi nút, khóa của nút đang xét lớn hơn khóa của tất cả các nút thuộc cây con trái và nhỏ hơn khóa của tất cả các nút thuộc cây con phải.

Dưới đây là một ví dụ về cây nhị phân tìm kiếm:



Nhờ ràng buộc về khóa trên cây nhị phân tìm kiếm, việc tìm kiếm trở nên có định hướng. Hơn nữa, do cấu trúc cây việc tìm kiếm trở nên nhanh đáng kể. Nếu số nút trên cây là N thì chi phí tìm kiếm trung bình chỉ khoảng $\log_2 N$.

Trong thực tế, khi xét đến cây nhị phân chủ yếu người ta xét cây nhị phân tìm kiếm.

6.2. CÁC THAO TÁC TRÊN CÂY NHỊ PHÂN TÌM KIẾM

6.2.1. Duyệt cây

Thao tác duyệt cây trên cây nhị phân tìm kiếm hoàn toàn giống như trên cây nhị phân. Chỉ có một lưu ý nhỏ là khi duyệt theo thứ tự giữa, trình tự các nút duyệt qua sẽ cho ta một dãy các nút theo thứ tự tăng dần của khóa.

6.2.2. Tìm một phần tử X trong cây

```
TNODE* searchNode(TREE T, Data X)
{
    if (T)
    {
        if (T->Key == X) return T;
        if (T->Key > X) return searchNode(T->pLeft, X);
        else return searchNode(T->pRight, X);
    }
    return NULL;
}
```

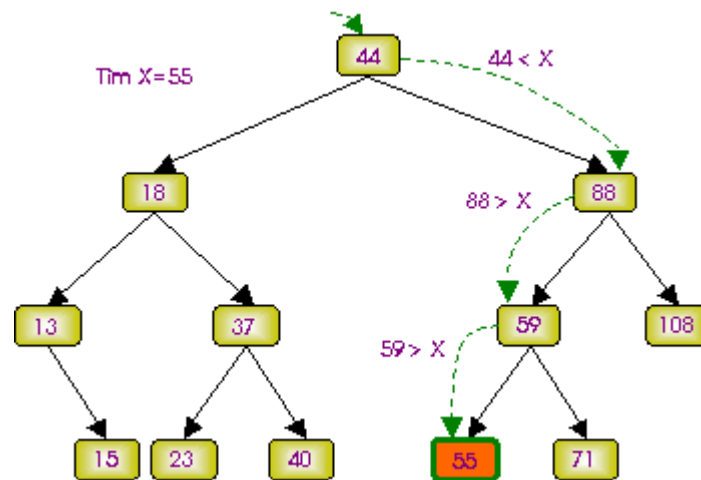
```
}
```

Ta có thể xây dựng một hàm tìm kiếm tương đương không đệ qui như sau:

```
TNODE * searchNode(TREE Root, Data x)
{
    NODE *p = Root;
    while (p != NULL)
    {
        if(x == p->Key) return p;
        else
            if(x < p->Key) p = p->pLeft;
            else p = p->pRight;
    }
    return NULL;
}
```

Dễ dàng thấy rằng số lần so sánh tối đa phải thực hiện để tìm phần tử X là h , với h là chiều cao của cây. Như vậy thao tác tìm kiếm trên cây nhị phân tìm kiếm có n nút tốn chi phí trung bình khoảng $O(\log_2 n)$.

Ví dụ: Tìm phần tử 55



6.2.3. Thêm một phần tử X vào cây

Việc thêm một phần tử X vào cây phải bảo đảm điều kiện ràng buộc của cây nhị phân tìm kiếm. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào một nút lá sẽ là tiện lợi nhất do ta có thể thực hiện quá trình tương tự thao tác tìm kiếm. Khi chấm dứt quá trình tìm kiếm cũng chính là lúc tìm được chỗ cần thêm.

Hàm insert trả về giá trị -1, 0, 1 khi không đủ bộ nhớ, gặp nút cũ hay thành công:

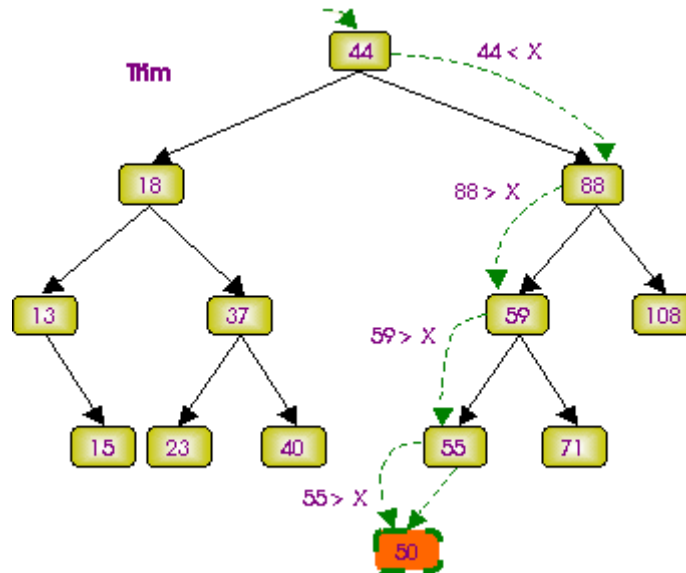
```
int insertNode(TREE &T, Data X)
{
    if (T)
    {
        if (T->Key == X)
            return 0; //đã có
        if (T->Key > X)
            return insertNode(T->pLeft, X);
        else
            return insertNode(T->pRight, X);
    }
}
```

```

T = new TNode;
if (T == NULL)
    return -1; //thiếu bộ nhớ
T->Key = X;
T->pLeft = T->pRight = NULL;
return 1; //thêm vào thành công
}

```

Ví dụ: Thêm phần tử 50



6.2.4. Hủy một phần tử có khóa X

Việc hủy một phần tử X ra khỏi cây phải bảo đảm điều kiện ràng buộc của cây nhị phân tìm kiếm.

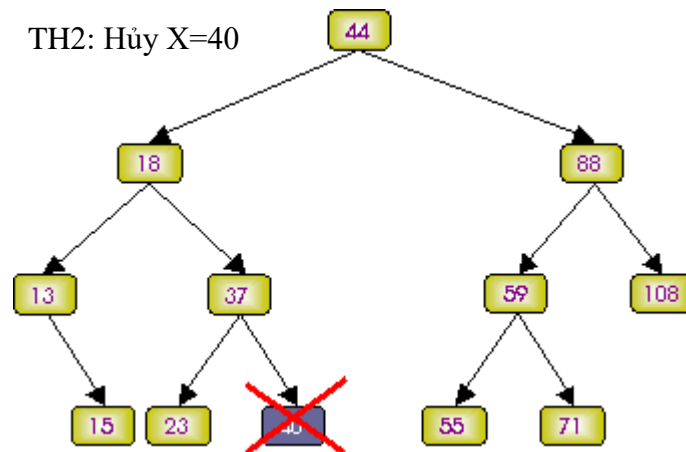
Có 3 trường hợp khi hủy nút X có thể xảy ra:

X là nút lá.

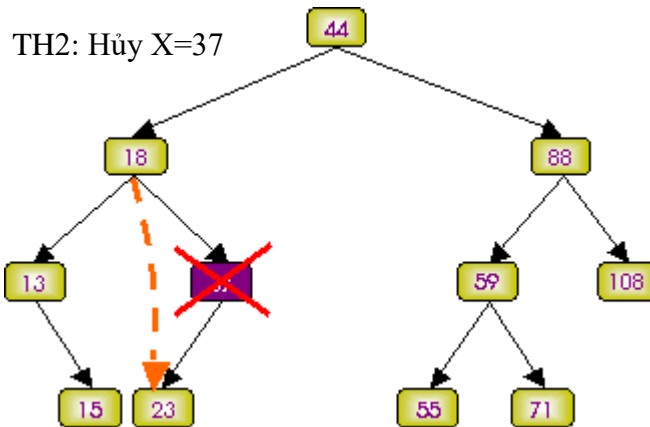
X chỉ có 1 con (trái hoặc phải).

X có đủ cả 2 con

Trường hợp thứ nhất: chỉ đơn giản hủy X vì nó không móc nối đến phần tử nào khác.



Trường hợp thứ hai: trước khi hủy X ta móc nối cha của X với con duy nhất của nó.



Trường hợp cuối cùng: ta không thể hủy trực tiếp do X có đủ 2 con \Rightarrow Ta sẽ hủy gián tiếp. Thay vì hủy X, ta sẽ tìm một phần tử thế mạng Y. Phần tử này có tối đa một con. Thông tin lưu tại Y sẽ được chuyển lên lưu tại X. Sau đó, nút bị hủy thật sự sẽ là Y giống như 2 trường hợp đầu.

Vấn đề là phải chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là cây nhị phân tìm kiếm.

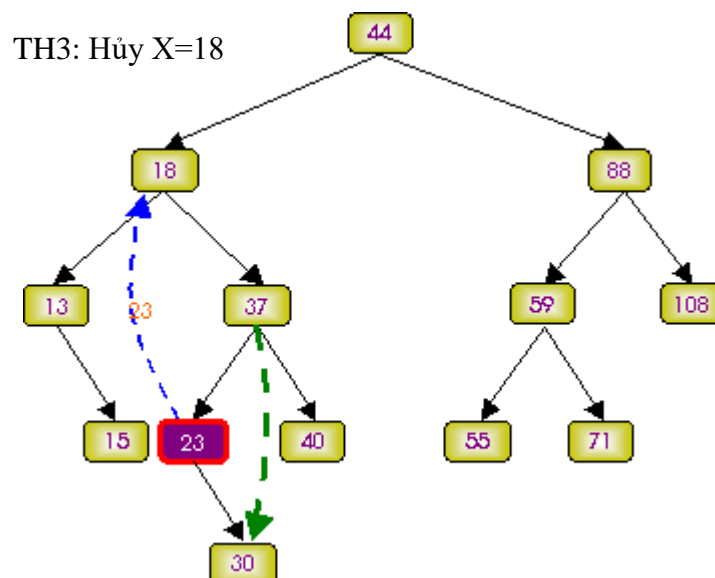
Có 2 phần tử thỏa mãn yêu cầu:

Phần tử nhỏ nhất (trái nhất) trên cây con phải.

Phần tử lớn nhất (phải nhất) trên cây con trái.

Việc chọn lựa phần tử nào là phần tử thế mạng hoàn toàn phụ thuộc vào ý thích của người lập trình. Ở đây, chúng tôi sẽ chọn phần tử (phải nhất trên cây con trái làm phần tử thế mạng.

Hãy xem ví dụ dưới đây để hiểu rõ hơn:



Sau khi hủy phần tử X=18 ra khỏi cây tình trạng của cây sẽ như trong hình dưới đây (phần tử 23 là phần tử thế mạng):

Hàm delNode trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây:


```

int delNode(TREE &T, Data X)
{
    if(T==NULL)    return 0;
    if(T->Key > X) return delNode (T->pLeft, X);
    if(T->Key < X) return delNode (T->pRight, X);
    else { //T->Key == X
        TNode*    p = T;
        if(T->pLeft == NULL)    T= T->pRight;
        else
            if(T->pRight == NULL)    T = T->pLeft;
            else {    //T có cả 2 con
                TNode*    q = T->pRight;
                searchStandFor(p, q);
            }
        delete p;
    }
}

```

Trong đó, hàm searchStandFor được viết như sau:

//Tìm phần tử thế mạng cho nút p

```

void searchStandFor(TREE &p, TREE &q)
{
    if(q->pLeft)    searchStandFor(p, q->pLeft);
    else {
        p->Key = q->Key;
        p = q;
        q = q->pRight;
    }
}

```

6.2.5. Tạo một cây nhị phân tìm kiếm

Ta có thể tạo một cây nhị phân tìm kiếm bằng cách lặp lại quá trình thêm 1 phần tử vào một cây rỗng.

6.2.6. Hủy toàn bộ cây nhị phân tìm kiếm

Việc toàn bộ cây có thể được thực hiện thông qua thao tác duyệt cây theo thứ tự sau. Nghĩa là ta sẽ hủy cây con trái, cây con phải rồi mới hủy nút gốc.

```

void removeTree(TREE &T)
{
    if(T)
    {
        removeTree(T->pLeft);
        removeTree(T->pRight);
        delete(T);
    }
}

```

Đánh giá:

Tất cả các thao tác searchNode, insertNode, delNode trên cây nhị phân tìm kiếm đều có độ phức tạp trung bình $O(h)$, với h là chiều cao của cây

Trong trường hợp tốt nhất, cây nhị phân tìm kiếm có n nút sẽ có độ cao $h = \log_2(n)$. Chi phí tìm kiếm khi đó sẽ tương đương tìm kiếm nhị phân trên mảng có thứ tự.

Tuy nhiên, trong trường hợp xấu nhất, cây có thể bị suy biến thành 1 DSLK (khi mà mỗi nút đều chỉ có 1 con trừ nút lá). Lúc đó các thao tác trên sẽ có độ phức tạp $O(n)$. Vì vậy cần có cải tiến cấu trúc của cây nhị phân tìm kiếm để đạt được chi phí cho các thao tác là $\log_2(n)$.

❖ **Bài tập củng cố:**

1. Hãy trình bày các vấn đề sau đây:

- Định nghĩa và đặc điểm của cây nhị phân tìm kiếm.
- Thao tác nào thực hiện tốt trong kiểu này.
- Hạn chế của kiểu này là gì ?

2. Xét thuật giải tạo cây nhị phân tìm kiếm. Nếu thứ tự các khóa nhập vào là như sau:

8 3 5 2 20 11 30 9 18 4

thì hình ảnh cây tạo được như thế nào ?

Sau đó, nếu hủy lần lượt các nút theo thứ tự như sau:

15, 20

thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ (nêu rõ phương pháp hủy khi nút có cả 2 cây con trái và phải)

3. Áp dụng thuật giải tạo cây nhị phân tìm kiếm cân bằng để tạo cây với thứ tự các khóa nhập vào là như sau:

5 7 2 1 3 6 10

thì hình ảnh cây tạo được như thế nào ? Giải thích rõ từng tình huống xảy ra khi thêm từng khóa vào cây và vẽ hình minh họa.

Sau đó, nếu hủy lần lượt các nút theo thứ tự như sau:

5, 6, 7, 10

thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ và giải thích

4. Giả sử A là một mảng các số thực đã có thứ tự tăng. Hãy viết hàm tạo một cây nhị phân tìm kiếm có chiều cao thấp nhất từ các phần tử của A.

5. Viết các hàm xác định các thông tin của cây nhị phân tìm kiếm:

- Số nút lá
- Số nút có đúng 1 cây con
- Số nút có đúng 2 cây con
- Số nút có khóa nhỏ hơn x
- Số nút có khóa lớn hơn x
- Số nút có khóa lớn hơn x và nhỏ hơn y
- Chiều cao của cây

CHƯƠNG 7

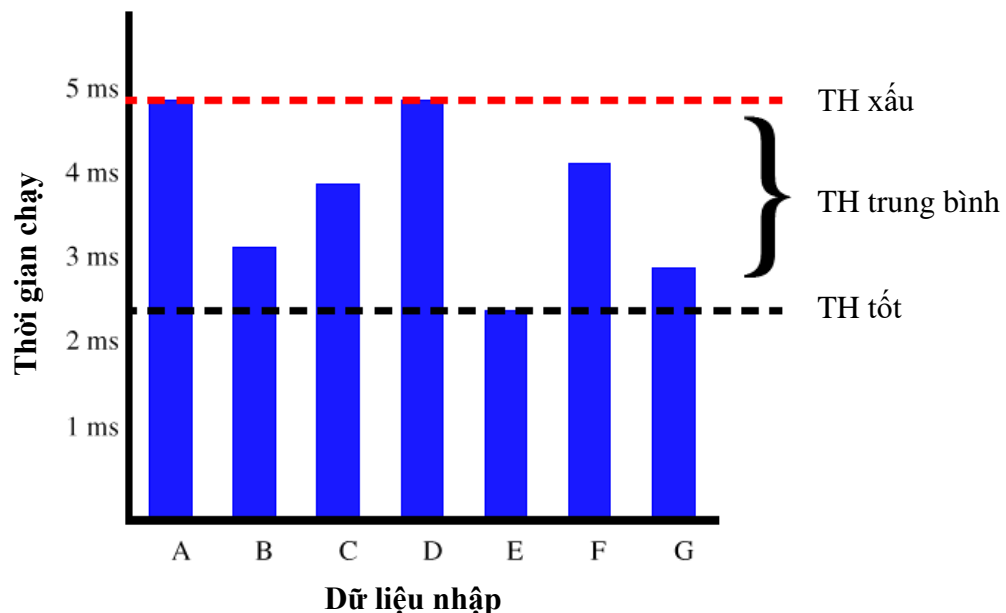
ĐÁNH GIÁ ĐỘ PHỨC TẠP THUẬT TOÁN

❖ **Mục tiêu học tập:** Sau khi học xong chương này, người học có thể:

- Hiểu được các bước phân tích thuật toán.
- Vận dụng và phân tích một số thuật toán cơ bản.
- Đánh giá được độ phức tạp của các thuật toán cơ bản.

Hầu hết các bài toán đều có nhiều thuật toán khác nhau để giải quyết chúng. Như vậy, làm thế nào để chọn được sự cài đặt tốt nhất? Đây là một lĩnh vực được phát triển tốt trong nghiên cứu về khoa học máy tính. Chúng ta sẽ thường xuyên có cơ hội tiếp xúc với các kết quả nghiên cứu mô tả các tính năng của các thuật toán cơ bản. Tuy nhiên, việc so sánh các thuật toán rất cần thiết và chắc chắn rằng một vài dòng hướng dẫn tổng quát về phân tích thuật toán sẽ rất hữu dụng.

Khi nói đến hiệu quả của một thuật toán, người ta thường quan tâm đến chi phí cần dùng để thực hiện nó. Chi phí này thể hiện qua việc sử dụng tài nguyên như bộ nhớ, thời gian sử dụng CPU, .. Ta có thể đánh giá thuật toán bằng phương pháp thực nghiệm thông qua việc cài đặt thuật toán rồi chọn các bộ dữ liệu thử nghiệm. Thống kê các thông số nhận được khi chạy các dữ liệu này ta sẽ có một đánh giá về thuật toán.



Tuy nhiên, phương pháp thực nghiệm có một số nhược điểm sau khiến cho nó khó có khả năng áp dụng trên thực tế:

Do phải cài đặt bằng một ngôn ngữ lập trình cụ thể nên thuật toán sẽ chịu sự hạn chế của ngôn ngữ lập trình này. Đồng thời, hiệu quả của thuật toán sẽ bị ảnh hưởng bởi trình độ của người cài đặt. Việc chọn được các bộ dữ liệu thử đặc trưng cho tất cả tập các dữ liệu vào của thuật toán là rất khó khăn và tốn nhiều chi phí. Các số liệu thu nhận được phụ thuộc nhiều vào phần cứng mà thuật toán được thử nghiệm trên đó. Điều này khiến cho việc so sánh các thuật toán khó khăn nếu chúng được thử nghiệm ở những nơi khác nhau.

Vì những lý do trên, người ta đã tìm kiếm những phương pháp đánh giá thuật toán hình thức hơn, ít phụ thuộc môi trường cũng như phần cứng hơn. Một phương pháp như vậy là phương pháp đánh giá thuật toán theo hướng xấp xỉ tiệm cận qua các khái niệm toán học O-lớn $O()$, o-nhỏ $o()$, $\Omega()$, $\Xi()$

Thông thường các vấn đề mà chúng ta giải quyết có một "kích thước" tự nhiên (thường là số lượng dữ liệu được xử lý) mà chúng ta sẽ gọi là N . Chúng ta muốn mô tả tài nguyên cần được dùng (thông thường nhất là thời gian cần thiết để giải quyết vấn đề) như một hàm số theo N . Chúng ta quan tâm đến trường hợp trung bình, tức là thời gian cần thiết để xử lý dữ liệu nhập thông thường, và cũng quan tâm đến trường hợp xấu nhất, tương ứng với thời gian cần thiết khi dữ liệu rơi vào trường hợp xấu nhất có thể có.

Việc xác định chi phí trong trường hợp trung bình thường được quan tâm nhiều nhất vì nó đại diện cho đa số trường hợp sử dụng thuật toán. Tuy nhiên, việc xác định chi phí trung bình này lại gặp nhiều khó khăn. Vì vậy, trong nhiều trường hợp, người ta xác định chi phí trong trường hợp xấu nhất (chặn trên) thay cho việc xác định chi phí trong trường hợp trung bình. Hơn nữa, trong một số bài toán, việc xác định chi phí trong trường hợp xấu nhất là rất quan trọng. Ví dụ, các bài toán trong hàng không, phẫu thuật.

7.1. CÁC BƯỚC PHÂN TÍCH THUẬT TOÁN

Bước đầu tiên trong việc phân tích một thuật toán là xác định đặc trưng dữ liệu sẽ được dùng làm dữ liệu nhập của thuật toán và quyết định phân tích nào là thích hợp. Về mặt lý tưởng, chúng ta muốn rằng với một phân bố tùy ý được cho của dữ liệu nhập, sẽ có sự phân bố tương ứng về thời gian hoạt động của thuật toán. Chúng ta không thể đạt tới điều lý tưởng này cho bất kỳ một thuật toán không tầm thường nào, vì vậy chúng ta chỉ quan tâm đến bao của thống kê về tính năng của thuật toán bằng cách cố gắng chứng minh thời gian chạy luôn luôn nhỏ hơn một "chặn trên" bất chấp dữ liệu nhập như thế nào và cố gắng tính được thời gian chạy trung bình cho dữ liệu nhập "ngẫu nhiên".

Bước thứ hai trong phân tích một thuật toán là nhận ra các thao tác trừu tượng của thuật toán để tách biệt sự phân tích với sự cài đặt. Ví dụ, chúng ta tách biệt sự nghiên cứu có bao nhiêu phép so sánh trong một thuật toán sắp xếp khỏi sự xác định cần bao nhiêu micro giây trên một máy tính cụ thể; yếu tố thứ nhất được xác định bởi tính chất của thuật toán, yếu tố thứ hai lại được xác định bởi tính chất của máy tính. Sự tách biệt này cho phép chúng ta so sánh các thuật toán một cách độc lập với sự cài đặt cụ thể hay độc lập với một máy tính cụ thể.

Bước thứ ba trong quá trình phân tích thuật toán là sự phân tích về mặt toán học, với mục đích tìm ra các giá trị trung bình và trường hợp xấu nhất cho mỗi đại lượng cơ bản. Chúng ta sẽ không gặp khó khăn khi tìm một chặn trên cho thời gian chạy chương trình, vấn đề ở chỗ là phải tìm ra chặn trên tốt nhất, tức là thời gian chạy chương trình khi gặp dữ liệu nhập của trường hợp xấu nhất. Trường hợp trung bình thông thường đòi hỏi một phân tích toán học tinh vi hơn trường hợp xấu nhất. Mỗi khi đã hoàn thành một quá trình phân tích thuật toán dựa vào các đại lượng cơ bản, nếu thời gian kết hợp với mỗi đại lượng được xác định rõ thì ta sẽ có các biểu thức để tính thời gian chạy.

Nói chung, tính năng của một thuật toán thường có thể được phân tích ở một mức độ vô cùng chính xác, chỉ bị giới hạn bởi tính năng không chắc chắn của máy tính hay bởi sự khó khăn trong việc xác định các tính chất toán học của một vài đại lượng trừu tượng. Tuy nhiên, thay vì phân tích một cách chi tiết chúng ta thường thích ước lượng để tránh sa vào chi tiết.

7.2. SỰ PHÂN LỚP CÁC THUẬT TOÁN

Như đã được chú ý trong ở trên, hầu hết các thuật toán đều có một tham số chính là N , thông thường đó là số lượng các phần tử dữ liệu được xử lý mà ảnh hưởng rất nhiều tới thời gian chạy. Tham số N có thể là bậc của một đa thức, kích thước của một tập tin được sắp xếp hay tìm kiếm, số nút trong một đồ thị .v.v... Hầu hết tất cả các thuật toán trong giáo trình này có thời gian chạy tiệm cận tới một trong các hàm sau:

7.2.1. Hằng số:

Hầu hết các chỉ thị của các chương trình đều được thực hiện một lần hay nhiều nhất chỉ

một vài lần. Nếu tất cả các chỉ thị của cùng một chương trình có tính chất này thì chúng ta sẽ nói rằng thời gian chạy của nó là hằng số. Điều này hiển nhiên là hoàn cảnh phân đầu để đạt được trong việc thiết kế thuật toán.

7.2.2. Hàm $\log N$:

Khi thời gian chạy của chương trình là logarit tức là thời gian chạy chương trình tiến chậm khi N lớn dần. Thời gian chạy thuộc loại này xuất hiện trong các chương trình mà giải một bài toán lớn bằng cách chuyển nó thành một bài toán nhỏ hơn, bằng cách cắt bỏ kích thước bớt một hằng số nào đó. Với mục đích của chúng ta, thời gian chạy có được xem như nhỏ hơn một hằng số "lớn". Cơ sở của logarit làm thay đổi hằng số đó nhưng không nhiều: khi N là một ngàn thì $\log N$ là 3 nếu cơ sở là 10, là 10 nếu cơ sở là 2; khi N là một triệu, $\log N$ được nhân gấp đôi. Bất cứ khi nào N được nhân đôi, $\log N$ tăng lên thêm một hằng số, nhưng $\log N$ không bị nhân gấp đôi khi N tăng tới N^2 .

7.2.3. Hàm N :

Khi thời gian chạy của một chương trình là tuyến tính, nói chung đây trường hợp mà một số lượng nhỏ các xử lý được làm cho mỗi phần tử dữ liệu nhập. Khi N là một triệu thì thời gian chạy cũng cỡ như vậy. Khi N được nhân gấp đôi thì thời gian chạy cũng được nhân gấp đôi. Đây là tình huống tối ưu cho một thuật toán mà phải xử lý N dữ liệu nhập (hay sản sinh ra N dữ liệu xuất).

7.2.4. Hàm $N \log N$:

Đây là thời gian chạy tăng dần lên cho các thuật toán mà giải một bài toán bằng cách tách nó thành các bài toán con nhỏ hơn, kể đến giải quyết chúng một cách độc lập và sau đó tổ hợp các lời giải. Bởi vì thiếu một tính từ tốt hơn (có lẽ là "tuyến tính logarit"?), chúng ta nói rằng thời gian chạy của thuật toán như thế là " $N \log N$ ". Khi N là một triệu, $N \log N$ có lẽ khoảng hai mươi triệu. Khi N được nhân gấp đôi, thời gian chạy bị nhân lên nhiều hơn gấp đôi (nhưng không nhiều lắm).

7.2.5. Hàm N^2 :

Khi thời gian chạy của một thuật toán là bậc hai, trường hợp này chỉ có ý nghĩa thực tế cho các bài toán tương đối nhỏ. Thời gian bình phương thường tăng dần lên trong các thuật toán mà xử lý tất cả các cặp phần tử dữ liệu (có thể là hai vòng lặp lồng nhau). Khi N là một ngàn thì thời gian chạy là một triệu. Khi N được nhân đôi thì thời gian chạy tăng lên gấp bốn lần.

7.2.6. Hàm N^3 :

Tương tự, một thuật toán mà xử lý các bộ ba của các phần tử dữ liệu (có lẽ là ba vòng lặp lồng nhau) có thời gian chạy bậc ba và cũng chỉ có ý nghĩa thực tế trong các bài toán nhỏ. Khi N là một trăm thì thời gian chạy là một triệu. Khi N được nhân đôi, thời gian chạy tăng lên gấp tám lần.

7.2.7. Hàm $2N$:

Một số ít thuật toán có thời gian chạy lũy thừa lại thích hợp trong một số trường hợp thực tế, mặc dù các thuật toán như thế là "sự ép buộc thô bạo" để giải các bài toán. Khi N là hai mươi thì thời gian chạy là một triệu. Khi N gấp đôi thì thời gian chạy được nâng lên lũy thừa hai!

Thời gian chạy của một chương trình cụ thể đôi khi là một hệ số hằng nhân với các số hạng nói trên ("số hạng dẫn đầu") cộng thêm một số hạng nhỏ hơn. Giá trị của hệ số hằng và các số hạng phụ thuộc vào kết quả của sự phân tích và các chi tiết cài đặt. Hệ số của số hạng dẫn đầu liên quan tới số chỉ thị bên trong vòng lặp: ở một tầng tùy ý của thiết kế thuật toán thì phải cẩn thận giới hạn số chỉ thị như thế. Với N lớn thì các số hạng dẫn đầu đóng vai trò chủ chốt; với N nhỏ thì các số hạng cùng đóng góp vào và sự so sánh các thuật toán sẽ khó khăn hơn. Trong hầu hết các trường hợp, chúng ta sẽ gặp các chương trình có thời gian chạy là

"tuyến tính", " $N \log N$ ", "bậc ba", ... với hiểu ngầm là các phân tích hay nghiên cứu thực tế phải được làm trong trường hợp mà tính hiệu quả là rất quan trọng.

7.3. PHÂN TÍCH TRƯỜNG HỢP TRUNG BÌNH

Một tiếp cận trong việc nghiên cứu tính năng của thuật toán là khảo sát trường hợp trung bình. Trong tình huống đơn giản nhất, chúng ta có thể đặc trưng chính xác các dữ liệu nhập của thuật toán: ví dụ một thuật toán sắp xếp có thể thao tác trên một mảng N số nguyên ngẫu nhiên, hay một thuật toán hình học có thể xử lý N điểm ngẫu nhiên trên mặt phẳng với các tọa độ nằm giữa 0 và 1. Kế đến là tính toán thời gian thực hiện trung bình của mỗi chỉ thị, và tính thời gian chạy trung bình của chương trình bằng cách nhân tần số sử dụng của mỗi chỉ thị với thời gian cần cho chỉ thị đó, sau cùng cộng tất cả chúng với nhau. Tuy nhiên có ít nhất ba khó khăn trong cách tiếp cận này như thảo luận dưới đây.

Trước tiên là trên một số máy tính rất khó xác định chính xác số lượng thời gian đòi hỏi cho mỗi chỉ thị. Trường hợp xấu nhất thì đại lượng này bị thay đổi và một số lượng lớn các phân tích chi tiết cho một máy tính có thể không thích hợp đối với một máy tính khác. Đây chính là vấn đề mà các nghiên cứu về độ phức tạp tính toán cũng cần phải né tránh.

Thứ hai, chính việc phân tích trường hợp trung bình lại thường là đòi hỏi toán học quá khó. Do tính chất tự nhiên của toán học thì việc chứng minh các chặn trên thì thường ít phức tạp hơn bởi vì không cần sự chính xác. Hiện nay chúng ta chưa biết được tính năng trong trường hợp trung bình của rất nhiều thuật toán.

Thứ ba (và chính là điều quan trọng nhất) trong việc phân tích trường hợp trung bình là mô hình dữ liệu nhập có thể không đặc trưng đầy đủ dữ liệu nhập mà chúng ta gặp trong thực tế. Ví dụ như làm thế nào để đặc trưng được dữ liệu nhập cho chương trình xử lý văn bản tiếng Anh? Một tác giả đề nghị nên dùng các mô hình dữ liệu nhập chẳng hạn như "tập tin thứ tự ngẫu nhiên" cho thuật toán sắp xếp, hay "tập hợp điểm ngẫu nhiên" cho thuật toán hình học, đối với những mô hình như thế thì có thể đạt được các kết quả toán học mà tiên đoán được tính năng của các chương trình chạy trên các ứng dụng thông thường.

7.4. SỰ CẦN THIẾT PHẢI PHÂN TÍCH GIẢI THUẬT

Trong khi giải một bài toán chúng ta có thể có một số giải thuật khác nhau, vấn đề là cần phải đánh giá các giải thuật đó để lựa chọn một giải thuật tốt (nhất). Thông thường thì ta sẽ căn cứ vào các tiêu chuẩn sau:

1. Giải thuật đúng đắn.
2. Giải thuật đơn giản.
3. Giải thuật thực hiện nhanh.

Với yêu cầu, để kiểm tra tính đúng đắn của giải thuật chúng ta có thể cài đặt giải thuật đó và cho thực hiện trên máy với một số bộ dữ liệu mẫu rồi lấy kết quả thu được so sánh với kết quả đã biết. Thực ra thì cách làm này không chắc chắn bởi vì có thể giải thuật đúng với tất cả các bộ dữ liệu chúng ta đã thử nhưng lại sai với một bộ dữ liệu nào đó. Và lại cách làm này chỉ phát hiện ra giải thuật sai chứ chưa chứng minh được là nó đúng. Tính đúng đắn của giải thuật cần phải được chứng minh bằng toán học. Tất nhiên điều này không đơn giản và do vậy chúng ta sẽ không đề cập đến ở đây.

Khi chúng ta viết một chương trình để sử dụng một vài lần thì yêu cầu là quan trọng nhất. Chúng ta cần một giải thuật để viết chương trình để nhanh chóng có được kết quả, thời gian thực hiện chương trình không được đề cao vì dù sao thì chương trình đó cũng chỉ sử dụng một vài lần mà thôi.

Tuy nhiên khi một chương trình được sử dụng nhiều lần thì yêu cầu tiết kiệm thời gian

thực hiện chương trình lại rất quan trọng đặc biệt đối với những chương trình mà khi thực hiện cần dữ liệu nhập lớn do đó yêu cầu sẽ được xem xét một cách kỹ càng. Ta gọi nó là hiệu quả thời gian thực hiện của giải thuật.

7.5. THỜI GIAN THỰC HIỆN CỦA CHƯƠNG TRÌNH

Một phương pháp để xác định hiệu quả thời gian thực hiện của một giải thuật là lập trình nó và đo lường thời gian thực hiện của hoạt động trên một máy tính xác định đối với tập hợp được chọn lọc các dữ liệu vào.

Thời gian thực hiện không chỉ phụ thuộc vào giải thuật mà còn phụ thuộc vào tập các chỉ thị của máy tính, chất lượng của máy tính và kỹ xảo của người lập trình. Sự thi hành cũng có thể điều chỉnh để thực hiện tốt trên tập đặc biệt các dữ liệu vào được chọn. Để vượt qua các trở ngại này, các nhà khoa học máy tính đã chấp nhận tính phức tạp của thời gian được tiếp cận như một sự đo lường cơ bản sự thực thi của giải thuật. Thuật ngữ tính hiệu quả sẽ đề cập đến sự đo lường này và đặc biệt đối với sự phức tạp thời gian trong trường hợp xấu nhất.

7.5.1. Thời gian thực hiện chương trình.

Thời gian thực hiện một chương trình là một hàm của kích thước dữ liệu vào, ký hiệu $T(n)$ trong đó n là kích thước (độ lớn) của dữ liệu vào.

Ví dụ: Chương trình tính tổng của n số có thời gian thực hiện là $T(n) = cn$ trong đó c là một hằng số.

Thời gian thực hiện chương trình là một hàm không âm, tức là $T(n) \geq 0$ với $n \geq 0$.

7.5.2. Đơn vị đo thời gian thực hiện.

Đơn vị của $T(n)$ không phải là đơn vị đo thời gian bình thường như giờ, phút giây... mà thường được xác định bởi số các lệnh được thực hiện trong một máy tính lý tưởng.

Ví dụ: Khi ta nói thời gian thực hiện của một chương trình là $T(n) = Cn$ thì có nghĩa là chương trình ấy cần Cn chỉ thị thực thi.

7.5.3. Thời gian thực hiện trong trường hợp xấu nhất.

Nói chung thì thời gian thực hiện chương trình không chỉ phụ thuộc vào kích thước mà còn phụ thuộc vào tính chất của dữ liệu vào. Nghĩa là dữ liệu vào có cùng kích thước nhưng thời gian thực hiện chương trình có thể khác nhau. Chẳng hạn chương trình sắp xếp dãy số nguyên tăng dần, khi ta cho vào dãy có thứ tự thì thời gian thực hiện khác với khi ta cho vào dãy chưa có thứ tự, hoặc khi ta cho vào một dãy đã có thứ tự tăng thì thời gian thực hiện cũng khác so với khi ta cho vào một dãy đã có thứ tự giảm.

Vì vậy thường ta coi $T(n)$ là thời gian thực hiện chương trình trong trường hợp xấu nhất trên dữ liệu vào có kích thước n , tức là: $T(n)$ là thời gian lớn nhất để thực hiện chương trình đối với mọi dữ liệu vào có cùng kích thước n .

7.6. TỶ SUẤT TĂNG VÀ ĐỘ PHỨC TẠP CỦA GIẢI THUẬT

7.6.1. Tỷ suất tăng

Ta nói rằng hàm không âm $T(n)$ có tỷ suất tăng (growth rate) $f(n)$ nếu tồn tại các hằng số C và N_0 sao cho $T(n) \leq Cf(n)$ với mọi $n \geq N_0$.

Ta có thể chứng minh được rằng “Cho một hàm không âm $T(n)$ bất kỳ, ta luôn tìm được tỷ suất tăng $f(n)$ của nó”.

Ví dụ: Giả sử $T(0) = 1$, $T(1) = 4$ và tổng quát $T(n) = (n+1)^2$. Đặt $N_0 = 1$ và $C=4$ thì với mọi $n \geq 1$ chúng ta dễ dàng chứng minh được rằng $T(n) = (n+1)^2 \leq 4n^2$ với mọi $n \geq 1$, tức là tỷ suất tăng của $T(n)$ là n^2 .

Ví dụ: Tỷ suất tăng của hàm $T(n) = 3n^3 + 2n^2$ là n^3 . Thực vậy, cho $N_0 = 0$ và $C = 5$ ta dễ dàng chứng minh rằng với mọi $n \geq 0$ thì $3n^3 + 2n^2 \leq 5n^3$

7.6.2. Khái niệm độ phức tạp của giải thuật

Giả sử ta có hai giải thuật P1 và P2 với thời gian thực hiện tương ứng là $T_1(n) = 100n^2$ (với tỷ suất tăng là n^2) và $T_2(n) = 5n^3$ (với tỷ suất tăng là n^3). Giải thuật nào sẽ thực hiện nhanh hơn? Câu trả lời phụ thuộc vào kích thước dữ liệu vào. Với $n < 20$ thì P2 sẽ nhanh hơn P1 ($T_2 < T_1$), do hệ số của $5n^3$ nhỏ hơn hệ số của $100n^2$ ($5 < 100$). Nhưng khi $n > 20$ thì ngược lại do số mũ của $100n^2$ nhỏ hơn số mũ của $5n^3$ ($2 < 3$). Ở đây chúng ta chỉ nên quan tâm đến trường hợp $n > 20$ vì khi $n < 20$ thì thời gian thực hiện của cả P1 và P2 đều không lớn và sự khác biệt giữa T_1 và T_2 là không đáng kể.

Như vậy một cách hợp lý là ta xét tỷ suất tăng của hàm thời gian thực hiện chương trình thay vì xét chính bản thân thời gian thực hiện.

Cho một hàm $T(n)$, $T(n)$ gọi là có độ phức tạp $f(n)$ nếu tồn tại các hằng C, N_0 sao cho $T(n) \leq C f(n)$ với mọi $n \geq N_0$ (tức là $T(n)$ có tỷ suất tăng là $f(n)$) và kí hiệu $T(n)$ là $O(f(n))$ (đọc là “ô của $f(n)$ ”)

Ví dụ: $T(n) = (n+1)^2$ có tỷ suất tăng là n^2 nên $T(n) = (n+1)^2$ là $O(n^2)$

Chú ý: $O(C.f(n)) = O(f(n))$ với C là hằng số. Đặc biệt $O(C) = O(1)$

Nói cách khác độ phức tạp tính toán của giải thuật là một hàm chặn trên của hàm thời gian. Vì hằng nhân tử C trong hàm chặn trên không có ý nghĩa nên ta có thể bỏ qua vì vậy hàm thể hiện độ phức tạp có các dạng thường gặp sau: $\log_2 n, n, n \log_2 n, n^2, n^3, 2^n, n!, n^n$. Ba hàm cuối cùng ta gọi là dạng hàm mũ, các hàm khác gọi là hàm đa thức. Một giải thuật mà thời gian thực hiện có độ phức tạp là một hàm đa thức thì chấp nhận được tức là có thể cài đặt để thực hiện, còn các giải thuật có độ phức tạp hàm mũ thì phải tìm cách cải tiến giải thuật.

Vì ký hiệu $\log_2 n$ thường có mặt trong độ phức tạp nên trong khuôn khổ tài liệu này, ta sẽ dùng **logn** thay thế cho **log₂n** với mục đích duy nhất là để cho gọn trong cách viết.

Khi nói đến độ phức tạp của giải thuật là ta muốn nói đến hiệu quả của thời gian thực hiện của chương trình nên ta có thể xem việc xác định thời gian thực hiện của chương trình chính là xác định độ phức tạp của giải thuật.

7.7. CÁCH TÍNH ĐỘ PHỨC TẠP

Cách tính độ phức tạp của một giải thuật bất kỳ là một vấn đề không đơn giản. Tuy nhiên ta có thể tuân theo một số nguyên tắc sau:

7.7.1. Quy tắc cộng

Nếu $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P1 và P2; và $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ thì thời gian thực hiện của đoạn hai chương trình đó nối tiếp nhau là $T(n) = O(\max(f(n), g(n)))$

Ví dụ: Lệnh gán $x := 15$ tốn một hằng thời gian hay $O(1)$, Lệnh đọc dữ liệu $READ(x)$ tốn một hằng thời gian hay $O(1)$. Vậy thời gian thực hiện cả hai lệnh trên nối tiếp nhau là $O(\max(1, 1)) = O(1)$

7.7.2. Quy tắc nhân

Nếu $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P1 và P2 và $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ thì thời gian thực hiện của đoạn hai đoạn chương trình đó lồng nhau là $T(n) = O(f(n).g(n))$

7.7.3. Quy tắc tổng quát để phân tích một chương trình:

Thời gian thực hiện của mỗi lệnh gán, READ, WRITE là $O(1)$

Thời gian thực hiện của một chuỗi tuần tự các lệnh được xác định bằng quy tắc cộng. Như vậy thời gian này là thời gian thi hành một lệnh nào đó lâu nhất trong chuỗi lệnh.

Thời gian thực hiện cấu trúc IF là thời gian lớn nhất thực hiện lệnh sau THEN hoặc sau ELSE và thời gian kiểm tra điều kiện. Thường thời gian kiểm tra điều kiện là $O(1)$.

Thời gian thực hiện vòng lặp là tổng (trên tất cả các lần lặp) thời gian thực hiện thân vòng lặp. Nếu thời gian thực hiện thân vòng lặp không đổi thì thời gian thực hiện vòng lặp là tích của số lần lặp với thời gian thực hiện thân vòng lặp.

Ví dụ: Tính thời gian thực hiện của thủ tục sắp xếp “nổi bọt”

```
void BubbleSort(int a[n])
{
    int i, j, temp;
    for(i= 0; i<=n-2; i++) /*1*/
        for(j=n-1; j>=i+1; j--) /*2*/
            if (a[j].key < a[j-1].key) /*3*/
            {
                temp=a[j-1]; /*4*/
                a[j-1] = a[j]; /*5*/
                a[j] = temp; /*6*/
            }
}
```

Ở đây, chúng ta chỉ quan tâm đến độ phức tạp của giải thuật.

Ta thấy toàn bộ chương trình chỉ gồm một lệnh lặp {1}, lồng trong lệnh {1} là lệnh {2}, lồng trong lệnh {2} là lệnh {3} và lồng trong lệnh {3} là 3 lệnh nối tiếp nhau {4}, {5} và {6}. Chúng ta sẽ tiến hành tính độ phức tạp theo thứ tự từ trong ra.

Trước hết, cả ba lệnh gán {4}, {5} và {6} đều tốn $O(1)$ thời gian, việc so sánh $a[j-1] > a[j]$ cũng tốn $O(1)$ thời gian, do đó lệnh {3} tốn $O(1)$ thời gian.

Vòng lặp {2} thực hiện $(n-i)$ lần, mỗi lần $O(1)$ do đó vòng lặp {2} tốn $O((n-i).1) = O(n-i)$. Vòng lặp {1} lặp có i chạy từ 1 đến $n-1$ nên thời gian thực hiện của vòng lặp {1} và cũng là độ phức tạp của giải thuật là

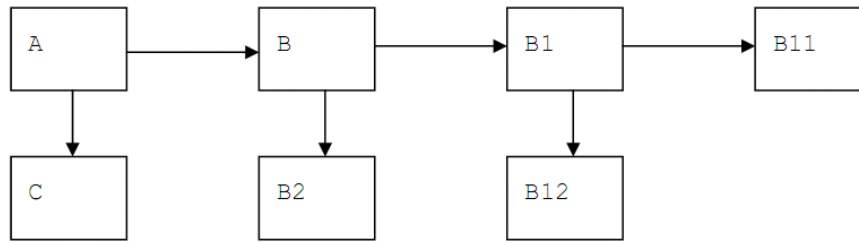
$$T(n) = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} = O(n^2).$$

Chú ý: Trong trường hợp vòng lặp không xác định được số lần lặp thì chúng ta phải lấy số lần lặp trong trường hợp xấu nhất.

7.7.4. Độ phức tạp của chương trình có gọi chương trình con không đệ quy

Nếu chúng ta có một chương trình với các chương trình con không đệ quy, để tính thời gian thực hiện của chương trình, trước hết chúng ta tính thời gian thực hiện của các chương trình con không gọi các chương trình con khác. Sau đó chúng ta tính thời gian thực hiện của các chương trình con chỉ gọi các chương trình con mà thời gian thực hiện của chúng đã được tính. Chúng ta tiếp tục quá trình đánh giá thời gian thực hiện của mỗi chương trình con sau khi thời gian thực hiện của tất cả các chương trình con mà nó gọi đã được đánh giá. Cuối cùng ta tính thời gian cho chương trình chính.

Giả sử ta có một hệ thống các chương trình gọi nhau theo sơ đồ sau:



Sơ đồ gọi thực hiện các chương trình con không đệ quy

Chương trình A gọi hai chương trình con là B và C, chương trình B gọi hai chương trình con là B1 và B2, chương trình B1 gọi hai chương trình con là B11 và B12.

Để tính thời gian thực hiện của A, ta tính theo các bước sau:

1. Tính thời gian thực hiện của C, B2, B11 và B12. Vì các chương trình con này không gọi chương trình con nào cả.
2. Tính thời gian thực hiện của B1. Vì B1 gọi B11 và B12 mà thời gian thực hiện của B11 và B12 đã được tính ở bước 1.
3. Tính thời gian thực hiện của B. Vì B gọi B1 và B2 mà thời gian thực hiện của B1 đã được tính ở bước 2 và thời gian thực hiện của B2 đã được tính ở bước 1.
4. Tính thời gian thực hiện của A. Vì A gọi B và C mà thời gian thực hiện của B đã được tính ở bước 3 và thời gian thực hiện của C đã được tính ở bước 1.

Ví dụ: Ta có thể viết lại chương trình sắp xếp bubble như sau: Trước hết chúng ta viết thủ tục Swap để thực hiện việc hoán đổi hai phần tử cho nhau, sau đó trong thủ tục Bubble, khi cần ta sẽ gọi đến thủ tục Swap này

```

void Swap(int &x, int &y)
{
    int t=x;
    x=y;
    y=t;
}

void BubbleSort(int a[n])
{
    int i, j, temp;
    for(i= 0; i<=n-2; i++) /*1*/
        for(j=n-1; j>=i+1; j--) /*2*/
            if (a[j].key < a[j-1].key)
                Swap(a[j-1], a[j]) /*3*/
}
  
```

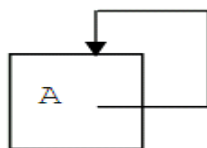
Trong cách viết trên, chương trình Bubble gọi chương trình con Swap, do đó để tính thời gian thực hiện của Bubble, trước hết ta cần tính thời gian thực hiện của Swap. Dễ thấy thời gian thực hiện của Swap là $O(1)$ vì nó chỉ bao gồm 3 lệnh gán.

Trong Bubble, lệnh {3} gọi Swap nên chỉ tốn $O(1)$, lệnh {2} thực hiện $n-i$ lần, mỗi lần tốn $O(1)$ nên tốn $O(n-i)$. Lệnh {1} thực hiện $n-1$ lần nên

$$T(n) = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} = O(n^2).$$

7.8. PHÂN TÍCH CÁC CHƯƠNG TRÌNH ĐỆ QUY

Với các chương trình có gọi các chương trình con đệ quy, ta không thể áp dụng cách tính như vừa trình bày trong mục 7.7.4 bởi vì một chương trình đệ quy sẽ gọi chính bản thân nó. Có thể thấy hình ảnh chương trình đệ quy A như sau:



Sơ đồ chương trình con A đệ quy

Với phương pháp tính độ phức tạp đã trình bày trong mục 7.7.4. thì không thể thực hiện được. Bởi vì nếu theo phương pháp đó thì, để tính thời gian thực hiện của chương trình A, ta phải tính thời gian thực hiện của chương trình A và cái vòng luẩn quẩn ấy không thể kết thúc được.

Với các chương trình đệ quy, trước hết ta cần thành lập các phương trình đệ quy, sau đó giải phương trình đệ quy, nghiệm của phương trình đệ quy sẽ là thời gian thực hiện của chương trình đệ quy.

7.8.1. Thành lập phương trình đệ quy

Phương trình đệ quy là một phương trình biểu diễn mối liên hệ giữa $T(n)$ và $T(k)$, trong đó $T(n)$ là thời gian thực hiện chương trình với kích thước dữ liệu nhập là n , $T(k)$ thời gian thực hiện chương trình với kích thước dữ liệu nhập là k , với $k < n$. Để thành lập được phương trình đệ quy, ta phải căn cứ vào chương trình đệ quy.

Thông thường một chương trình đệ quy để giải bài toán kích thước n , phải có ít nhất một trường hợp dừng ứng với một n cụ thể và lời gọi đệ quy để giải bài toán kích thước k ($k < n$).

Để thành lập phương trình đệ quy, ta gọi $T(n)$ là thời gian để giải bài toán kích thước n , ta có $T(k)$ là thời gian để giải bài toán kích thước k . Khi đệ quy dừng, ta phải xem xét khi đó chương trình làm gì và tốn hết bao nhiêu thời gian, chẳng hạn thời gian này là $c(n)$. Khi đệ quy chưa dừng thì phải xét xem có bao nhiêu lời gọi đệ quy với kích thước k ta sẽ có bấy nhiêu $T(k)$. Ngoài ra ta còn phải xem xét đến thời gian để phân chia bài toán và tổng hợp các lời giải, chẳng hạn thời gian này là $d(n)$.

Dạng tổng quát của một phương trình đệ quy sẽ là:

$$T(n) = \begin{cases} C(n) \\ F(T(k)) + d(n) \end{cases}$$

Trong đó $C(n)$ là thời gian thực hiện chương trình ứng với trường hợp đệ quy dừng.

$F(T(k))$ là một đa thức của các $T(k)$. $d(n)$ là thời gian để phân chia bài toán và tổng hợp các kết quả.

Ví dụ: Xét hàm tính giai thừa viết bằng giải thuật đệ quy như sau:

```
int Giaithua(int n)
{
    if (n==0) return 1;
    else return (n* Giaithua(n-1));
};
```

Gọi $T(n)$ là thời gian thực hiện việc tính n giai thừa, thì $T(n-1)$ là thời gian thực hiện việc tính $n-1$ giai thừa. Trong trường hợp $n = 0$ thì chương trình chỉ thực hiện return 1, nên tốn $O(1)$, do đó ta có $T(0) = C_1$. Trong trường hợp $n > 0$ chương trình phải gọi đệ quy $Giaithua(n-1)$, việc

gọi đệ quy này tốn $T(n-1)$, sau khi có kết quả của việc gọi đệ quy, chương trình phải nhân kết quả đó với n và gán cho $Giaithua$. Thời gian để thực hiện phép nhân và phép gán là một hằng C_2 . Vậy ta có

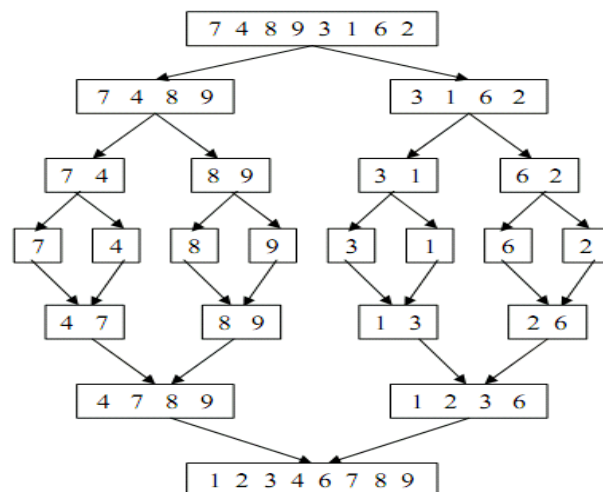
$$T(n) = \begin{cases} C_1 & (\text{Nếu } n=0) \\ T(n-1) + C_2 & (\text{Nếu } n>0) \end{cases}$$

Đây là phương trình đệ quy để tính thời gian thực hiện của chương trình đệ quy $Giaithua$.

Ví dụ: Chúng ta xét thủ tục MergeSort một cách phác thảo như sau:

```
List MergeSort (List L; int n)
{
    List L1, L2
    if (n==1) RETURN (L);
    else {
        Chia đôi L thành L1 và L2, với độ dài n/2;
        RETURN (Merge (MergeSort (L1, n/2), MergeSort (L2, n/2)));
    }
}
```

Chẳng hạn để sắp xếp danh sách L gồm 8 phần tử 7, 4, 8, 9, 3, 1, 6, 2 ta có mô hình minh họa của MergeSort như sau:



Hình minh họa sắp xếp trộn

Hàm MergeSort nhận một danh sách có độ dài n và trả về một danh sách đã được sắp xếp. Thủ tục Merge nhận hai danh sách đã được sắp $L1$ và $L2$ mỗi danh sách có độ dài $n/2$, trộn chúng lại với nhau để được một danh sách gồm n phần tử có thứ tự.

Gọi $T(n)$ là thời gian thực hiện MergeSort một danh sách n phần tử thì $T(n/2)$ là thời gian thực hiện MergeSort một danh sách $n/2$ phần tử.

Khi L có độ dài 1 ($n = 1$) thì chương trình chỉ làm một việc duy nhất là $\text{return}(L)$, việc này tốn $O(1) = C_1$ thời gian. Trong trường hợp $n > 1$, chương trình phải thực hiện gọi đệ quy MergeSort hai lần cho $L1$ và $L2$ với độ dài $n/2$ do đó thời gian để gọi hai lần đệ quy này là $2T(n/2)$. Ngoài ra còn phải tốn thời gian cho việc chia danh sách L thành hai nửa bằng nhau và trộn hai danh sách kết quả (Merge). Người ta xác định được thời gian để chia danh sách và Merge là $O(n) = C_2n$. Vậy ta có phương trình đệ quy như sau:

$$T(n) = \begin{cases} C_1 & \text{nếu } n = 1 \\ 2T(\frac{n}{2}) + C_2 n & \text{nếu } n > 1 \end{cases}$$

7.8.2. Giải phương trình đệ quy

Có ba phương pháp giải phương trình đệ quy:

1. Phương pháp truy hồi
2. Phương pháp đoán nghiệm.
3. Lời giải tổng quát của một lớp các phương trình đệ quy.

7.8.2.1. Phương pháp truy hồi

Dùng đệ quy để thay thế bất kỳ $T(m)$ với $m < n$ vào phía phải của phương trình cho đến khi tất cả $T(m)$ với $m > 1$ được thay thế bởi biểu thức của các $T(1)$ hoặc $T(0)$. Vì $T(1)$ và $T(0)$ luôn là hằng số nên chúng ta có công thức của $T(n)$ chứa các số hạng chỉ liên quan đến n và các hằng số. Từ công thức đó ta suy ra $T(n)$.

Ví dụ: Giải phương trình $T(n) = \begin{cases} C_1 & \text{nếu } n = 0 \\ T(n-1) + C_2 & \text{nếu } n > 0 \end{cases}$

Ta có $T(n) = T(n-1) + C_2$

$$T(n) = [T(n-2) + C_2] + C_2 = T(n-2) + 2C_2$$

$$T(n) = [T(n-3) + C_2] + 2C_2 = T(n-3) + 3C_2$$

.....

$$T(n) = T(n-i) + iC_2$$

Quá trình trên kết thúc khi $n - i = 0$ hay $i = n$. Khi đó ta có

$$T(n) = T(0) + nC_2 = C_1 + nC_2 = O(n)$$

Ví dụ: Giải phương trình $T(n) = \begin{cases} C_1 & \text{nếu } n = 1 \\ 2T(\frac{n}{2}) + C_2 n & \text{nếu } n > 1 \end{cases}$

Ta có $T(n) = 2T(\frac{n}{2}) + 2C_2 n$

$$T(n) = 2[2T(\frac{n}{4}) + C_2 \frac{n}{2}] + 2C_2 n = 4T(\frac{n}{4}) + 2C_2 n$$

$$T(n) = 4[2T(\frac{n}{8}) + C_2 \frac{n}{4}] + 2C_2 n = 8T(\frac{n}{8}) + 3C_2 n$$

.....

$$T(n) = 2^i T(\frac{n}{2^i}) + iC_2 n$$

Quá trình suy rộng sẽ kết thúc khi $\frac{n}{2^i} = 1$ hay $2^i = n$ và do đó $i = \log n$. Khi đó ta có:

$$T(n) = nT(1) + \log n C_2 n = C_1 n + C_2 n \log n = O(n \log n).$$

7.8.2.2. Lời giải tổng quát cho một lớp các phương trình đệ quy

Để giải một bài toán kích thước n , ta chia bài toán đã cho thành a bài toán con, mỗi bài toán con có kích thước n/b . Giải các bài toán con này và tổng hợp kết quả lại để được kết quả của

bài toán đã cho. Với các bài toán con chúng ta cũng sẽ áp dụng phương pháp đó để tiếp tục chia nhỏ ra nữa cho đến các bài toán con kích thước 1. Kỹ thuật này sẽ dẫn chúng ta đến một giải thuật đệ quy.

Giả thiết rằng mỗi bài toán con kích thước 1 lấy một đơn vị thời gian và thời gian để chia bài toán kích thước n thành các bài toán con kích thước n/b và tổng hợp kết quả từ các bài toán con để được lời giải của bài toán ban đầu là $d(n)$. (Chẳng hạn đối với ví dụ MergeSort, chúng ta có $a = b = 2$, và $d(n) = C_2n$. Xem C_1 là một đơn vị).

Tất cả các giải thuật đệ quy như trên đều có thể thành lập một phương trình đệ quy tổng quát, chung cho lớp các bài toán ấy.

Nếu gọi $T(n)$ là thời gian để giải bài toán kích thước n thì $T(n/b)$ là thời gian để giải bài toán con kích thước n/b . Khi $n = 1$ theo giả thiết trên thì thời gian giải bài toán kích thước 1 là 1 đơn vị, tức là $T(1) = 1$. Khi n lớn hơn 1, ta phải giải đệ quy a bài toán con kích thước n/b , mỗi bài toán con tốn $T(n/b)$ nên thời gian cho a lời giải đệ quy này là $aT(n/b)$. Ngoài ra ta còn phải tốn thời gian để phân chia bài toán và tổng hợp các kết quả, thời gian này theo giả thiết trên là $d(n)$. Vậy ta có phương trình đệ quy:

$$T(n) = \begin{cases} 1 & \text{Nếu } n=1 \\ aT(\frac{n}{b}) + d(n) & \text{Nếu } n>1 \end{cases} \quad (1)$$

Ta sử dụng phương pháp truy hồi để giải phương trình này. Khi $n > 1$ ta có

$$T(n) = aT(\frac{n}{b}) + d(n)$$

$$T(n) = a[aT(\frac{n}{b^2}) + d(\frac{n}{b})] + d(n) = a^2T(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n)$$

$$T(n) = a^2[aT(\frac{n}{b^3}) + d(\frac{n}{b^2})] + ad(\frac{n}{b}) + d(n) = a^3T(\frac{n}{b^3}) + a^2d(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n)$$

$$= \dots\dots$$

$$= a^i T(\frac{n}{b^i}) + \sum_{j=0}^{i-1} a^j d(\frac{n}{b^j})$$

Giả sử $n = b^k$, quá trình suy rộng trên sẽ kết thúc khi $i = k$.

Khi đó ta được $T(\frac{n}{b^k}) = T(1) = 1$. Thay vào trên ta có:

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}) \quad (2)$$

7.8.2.2.1. Hàm tiến triển, nghiệm thuần nhất và nghiệm riêng

Trong phương trình đệ quy (1) hàm thời gian $d(n)$ được gọi là **hàm tiến triển**.

Trong công thức (2), $a^k = n^{\log_b a}$ được gọi là **ngiem thuần nhất**.

Nghiệm thuần nhất là nghiệm chính xác khi $d(n) = 0$ với mọi n . Nói một cách khác, nghiệm thuần nhất biểu diễn thời gian để giải tất cả các bài toán con.

Trong công thức (2) $\sum_{j=0}^{k-1} a^j d(b^{k-j})$ được gọi là **nghiệm riêng**.

Nghiệm riêng biểu diễn thời gian phải tốn để tạo ra các bài toán con và tổng hợp các kết quả của chúng. Nhìn vào công thức ta thấy nghiệm riêng phụ thuộc vào hàm tiến triển, số lượng và kích thước các bài toán con.

Khi tìm nghiệm của phương trình (1), chúng ta phải tìm nghiệm riêng và so sánh với nghiệm thuần nhất. Nếu nghiệm nào lớn hơn, ta lấy nghiệm đó làm nghiệm của phương trình (1).

Việc xác định nghiệm riêng không đơn giản chút nào, tuy vậy, chúng ta cũng tìm được một lớp các hàm tiến triển có thể dễ dàng xác định nghiệm riêng.

7.8.2.2.2. Hàm nhân

Một hàm $f(n)$ được gọi là hàm nhân (multiplicative function) nếu $f(m.n) = f(m).f(n)$ với mọi số nguyên dương m và n .

Ví dụ: Hàm $f(n) = n^k$ là một hàm nhân, vì $f(m.n) = (m.n)^k = m^k.n^k = f(m).f(n)$

Tính nghiệm của phương trình tổng quát trong trường hợp $d(n)$ là hàm nhân:

Nếu $d(n)$ trong (1) là một hàm nhân thì theo tính chất của hàm nhân ta có $d(b^{k-j}) = [d(b)]^{k-j}$ và nghiệm riêng của (2) là

$$\sum_{j=0}^{k-1} a^j d(b^{k-j}) = \sum_{j=0}^{k-1} a^j [d(b)]^{k-j} = [d(b)]^k \sum_{j=0}^{k-1} \left[\frac{a}{d(b)} \right]^j = [d(b)]^k \frac{\left[\frac{a}{d(b)} \right]^k - 1}{\frac{a}{d(b)} - 1}$$

$$\text{Hay nghiệm riêng} = \frac{a^k - [d(b)]^k}{\frac{a}{d(b)} - 1} \quad (3)$$

Xét ba trường hợp sau:

1. **Trường hợp 1:** $a > d(b)$ thì trong công thức (3) ta có $a^k > [d(b)]^k$, theo quy tắc lấy độ phức tạp ta có nghiệm riêng là $O(a^k) = O(n^{\log_a a})$. Như vậy nghiệm riêng và nghiệm thuần nhất bằng nhau do đó **$T(n) = O(n^{\log_a a})$** .

Trong trường hợp này ta thấy thời gian thực hiện chỉ phụ thuộc vào a, b mà không phụ thuộc vào hàm tiến triển $d(n)$. Vì vậy **để cải tiến giải thuật ta cần giảm a hoặc tăng b** .

2. **Trường hợp 2:** $a < d(b)$ thì trong công thức (3) ta có $[d(b)]^k > a^k$, theo quy tắc lấy độ phức tạp ta có nghiệm riêng là $O([d(b)]^k) = O(n^{\log_b d(b)})$. Trong trường hợp này nghiệm riêng lớn hơn nghiệm thuần nhất nên **$T(n) = O(n^{\log_b d(b)})$** .

Để cải tiến giải thuật chúng ta cần giảm $d(b)$ hoặc tăng b .

Trường hợp đặc biệt quan trọng khi $d(n) = n$. Khi đó $d(b) = b$ và $\log_b b = 1$. Vì thế nghiệm riêng là $O(n)$ và do vậy **$T(n) = O(n)$** .

3. **Trường hợp 3:** $a = d(b)$ thì công thức (3) không xác định nên ta phải tính trực tiếp nghiệm riêng:

$$\text{Nghiệm riêng} = [d(b)]^k \sum_{j=0}^{k-1} \left[\frac{a}{d(b)} \right]^j = a^k \sum_{j=0}^{k-1} 1 = a^k k \quad (\text{do } a = d(b))$$

Do $n = b^k$ nên $k = \log_b n$ và $a^k = n^{\log_b a}$. Vậy nghiệm riêng là $n^{\log_b a} \log_b n$ và nghiệm này lớn gấp $\log_b n$ lần nghiệm thuần nhất. Do đó $T(n) = O(n^{\log_b a} \log_b n)$.

Chú ý khi giải một phương trình đệ quy cụ thể, ta phải xem phương trình đó có thuộc dạng phương trình tổng quát hay không. Nếu có thì phải xét xem hàm tiến triển có phải là hàm nhân không. Nếu có thì ta xác định a , $d(b)$ và dựa vào sự so sánh giữa a và $d(b)$ mà vận dụng một trong ba trường hợp nói trên.

Ví dụ: Giải các phương trình đệ quy sau với $T(1) = 1$ và

$$1/- T(n) = 4T\left(\frac{n}{2}\right) + n$$

$$2/- T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

$$3/- T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

Các phương trình đã cho đều có dạng phương trình tổng quát, các hàm tiến triển $d(n)$ đều là các hàm nhân và $a = 4$, $b = 2$.

Với phương trình thứ nhất, ta có $d(n) = n \Rightarrow d(b) = b = 2 < a$, áp dụng trường hợp 1 ta có $T(n) = O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)$.

Với phương trình thứ hai, $d(n) = n^2 \Rightarrow d(b) = b^2 = 4 = a$, áp dụng trường hợp 3 ta có $T(n) = O(n^{\log_b a} \log n) = O(n^{\log_2 4} \log n) = O(n^2 \log n)$.

Với phương trình thứ 3, ta có $d(n) = n^3 \Rightarrow d(b) = b^3 = 8 > a$, áp dụng trường hợp 2, ta có $T(n) = O(n^{\log_b d(b)}) = O(n^{\log_2 8}) = O(n^3)$.

7.8.2.2.3. Các hàm tiến triển khác

Trong trường hợp hàm tiến triển không phải là một hàm nhân thì chúng ta không thể áp dụng các công thức ứng với ba trường hợp nói trên mà chúng ta phải tính trực tiếp nghiệm riêng, sau đó so sánh với nghiệm thuần nhất để lấy nghiệm lớn nhất trong hai nghiệm đó làm nghiệm của phương trình.

Ví dụ: Giải phương trình đệ quy sau:

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n \log n$$

Phương trình đã cho thuộc dạng phương trình tổng quát nhưng $d(n) = n \log n$ không phải là một hàm nhân.

Ta có nghiệm thuần nhất $= n^{\log_b a} = n^{\log_2 2} = n$

Do $d(n) = n \log n$ không phải là hàm nhân nên ta phải tính nghiệm riêng bằng cách xét trực tiếp

$$\text{Nghiệm riêng} = \sum_{j=0}^{k-1} a^j d(b^{k-j}) = \sum_{j=0}^{k-1} 2^j 2^{k-j} \log 2^{k-j} = 2^k \sum_{j=0}^{k-1} (k-j) = 2^k \frac{k(k+1)}{2} = O(2^k k^2)$$

Theo giả thiết trong phương trình tổng quát thì $n = b^k$ nên $k = \log_b n$, ở đây do $b = 2$

nên $2^k = n$ và $k = \log n$, chúng ta có nghiệm riêng là $O(n \log^2 n)$, nghiệm này lớn hơn nghiệm thuần nhất do đó $T(n) = O(n \log^2 n)$.

❖ **Bài tập củng cố:**

1. Giải các phương trình đệ quy sau với $T(1) = 1$ và

a) $T(n) = 3T(n/2) + n$

b) $T(n) = 3T(n/2) + n^2$

c) $T(n) = 8T(n/2) + n^3$

2. Giải các phương trình đệ quy sau với $T(1) = 1$ và

a) $T(n) = 4T(n/3) + n$

b) $T(n) = 4T(n/3) + n^2$

c) $T(n) = 9T(n/3) + n^2$

3. Giải các phương trình đệ quy sau với $T(1) = 1$ và

a) $T(n) = T(n/2) + 1$

b) $T(n) = 2T(n/2) + \log n$

c) $T(n) = 2T(n/2) + n$

d) $T(n) = 2T(n/2) + n^2$

4. Cho một mảng n số nguyên được sắp thứ tự tăng. Viết hàm tìm một số nguyên trong mảng đó theo phương pháp **tìm kiếm nhị phân**, nếu tìm thấy thì trả về TRUE, ngược lại trả về FALSE. Sử dụng hai kỹ thuật là đệ quy và vòng lặp. Với mỗi kỹ thuật hãy viết một hàm tìm và tính thời gian thực hiện của hàm đó.

5. Tính thời gian thực hiện của giải thuật đệ quy giải bài toán Tháp Hà nội với n tầng?

6. Xét công thức truy toán để tính số tổ hợp chập k của n như sau:

$$C_n^k = \begin{cases} 1 & \text{Nếu } k=0 \text{ hoặc } k=n \\ C_{n-1}^{k-1} + C_{n-1}^k & \text{Nếu } 0 < k < n \end{cases}$$

a) Viết một hàm đệ quy để tính số tổ hợp chập k của n .

b) Tính thời gian thực hiện của giải thuật nói trên.

TÀI LIỆU THAM KHẢO

1. **A.V. Aho, J.E. Hopcroft, J.D. Ullman;** *Data Structures and Algorithms*; Addison- Wesley; 1983.
2. **Jeffrey H Kingston;** *Algorithms and Data Structures*; Addison-Wesley; 1998.
3. **Đinh Mạnh Tường;** *Cấu trúc dữ liệu & Thuật toán*; Nhà xuất bản khoa học và kỹ thuật; Hà nội-2001.
4. **Nguyễn Đức Nghĩa, Tô Văn Thành;** *Toán rời rạc*; 1997.
5. **Nguyễn Văn Linh,** *Giải thuật*, Đại học Cần Thơ, năm 2003.
6. **Trần Hạnh Nhi, Dương Anh Đức;** *Cấu trúc dữ liệu và giải thuật*, Đại học Khoa học Tự nhiên, TP.HCM